Towards Power- and Energy-Efficient Datacenters

by

Chang-Hong Hsu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Doctoral Committee:

Assistant Professor Jason Mars, Co-Chair
Assistant Professor Lingjia Tang, Co-Chair
Assistant Professor Hun-Seok Kim
Associate Professor Thomas F. Wenisch

Chang-Hong Hsu

hsuch@umich.edu

ORCID iD: 0000-0001-8107-122X

ACKNOWLEDGEMENTS

First and foremost I want to thank my advisers, Professor Lingjia Tang and Professor Jason Mars. You kindly welcomed me to the Clarity Lab when I feel directionless, and introduced me to an area full of excitement and challenges. During the past four years, you have helped me find where my interests are, learn how to establish hypotheses, and defend my works. Without your guidance, all of these would have been much more difficult. Lingjia, you make the strongest case for a rigorous researcher and a logical thinker. You have been and remains my best role model for a scientist and a mentor. Jason, you are one of the most encouraging and motivating people I have ever met. Your passion deeply impacts me and makes me always willing to take one step further to pursue the things I really want.

I must also express my gratitude to Professor Thomas Wenisch and Professor Hun-Seok Kim for their valuable discussion and guidance in constructing this dissertation.

Completing this work would have been much more difficult were it not with the support provided by the members of Clarity Lab – Mike, Yunqi, Animesh, Shih-Chieh, Austin, Parker, Ram, Matt, Yiping, Johann, Steve, Md, Vini, Hailong, Quan, Jeongseob, Arjun. The group has been a wonderful source of friendship as well as collaboration. It was through our countless brainstorming, debating, and discussion with you all that I grow and improve tremendously and become a better scientist.

Anna, my dear wife – thank you for your unconditional love and support that carry me through all the ups and downs throughout the years. It was you who guided me through the deepest darkness, and helped me realize that there will always to I

would not have been able to come to this far without you.

My parents, Hsin-Jung Lee and Chia-Chiao Hsu, and my dearest family members – thank you for always being by my side and respect my decisions. Thank you for always being thoughtful and caring about me and my well-being.

To my friend and all those who wished me well and who have helped me throughout this journey, thank you.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

As the Internet evolves, cloud computing is now a dominant form of computation in modern lives. Warehouse-scale computers (WSCs), or datacenters, comprising the foundation of this cloud-centric web have been able to deliver satisfactory performance to both the Internet companies and the customers. With the increased focus and popularity of the cloud, however, datacenter loads rise and grow rapidly, and Internet companies are in need of boosted computing capacity to serve such demand. Unfortunately, power and energy are often the major limiting factors prohibiting datacenter growth: it is often the case that no more servers can be added to datacenters without surpassing the capacity of the existing power infrastructure.

This dissertation aims to investigate the issues of power and energy usage in a modern datacenter environment. We identify the source of power and energy inefficiency at three levels in a modern datacenter environment and provides insights and solutions to address each of these problems, aiming to prepare datacenters for critical future growth. We start at the datacenter-level and find that the peak provisioning and improper service placement in multi-level power delivery infrastructures fragment the power budget inside production datacenters, degrading the compute capacity the existing infrastructure can support. We find that the heterogeneity among datacenter workloads is key to address this issue and design systematic methods to reduce the fragmentation and improve the utilization of the power budget. This dissertation then narrow the focus to examine the energy usage of individual servers running cloud workloads. Especially, we examine the power management mecha-

nisms employed in these servers and find that the coarse time granularity of these mechanisms is one critical factor that leads to excessive energy consumption. We propose an intelligent and low overhead solution on top of the emerging finer granularity voltage/frequency boosting circuit to effectively pinpoints and boosts queries that are likely to increase the tail distribution and can reap more benefit from the voltage/frequency boost, improving energy efficiency without sacrificing the quality of services. The final focus of this dissertation takes a further step to investigate how using a fundamentally more efficient computing substrate, field programmable gate arrays (FPGAs), benefit datacenter power and energy efficiency. Different from other types of hardware accelerations, FPGAs can be reconfigured on-the-fly to provide fine-grain control over hardware resource allocation and presents a unique set of challenges for optimal workload scheduling and resource allocation. We aim to design a set coordinated algorithms to manage these two key factors simultaneously and fully explore the benefit of deploying FPGAs in the highly varying cloud environment.

CHAPTER I

Introduction

With the evolution of the Internet, we have witnessed major advances, one of the most significant and recent being cloud computing. This increased focus on cloud computing has led to the rapid rise and growth of datacenter loads [2, 1]. Datacenter load is an important issue because it is directly related to datacenter profitability and customer satisfaction. A central issue in datacenter load is, with the power and energy supplied by the power delivery infrastructure in datacenters, how to manage the amount of computation that can be deployed to handle the ever growing load. Indeed, power has become one of the most contentious resources in datacenter management. Building new datacenters and power infrastructure would help accommodating the increased traffic, but the costly and time-consuming nature of this solution renders it infeasible. Another possibility is to address the power inefficiency that exists in datacenters and maximize the utilization of the existing power infrastructure. Research has shown that most datacenters suffer from power inefficiency[44, 150, 115, 54]. Fortunately, we are seeing new opportunity as new hardware and software technologies emerge[3, 74, 119, 121, 73]. It is worthwhile to investigate how these new technologies can be applied to help improve power efficiency.

This dissertation takes a top-down approach, presenting three works to examine

and address power and energy inefficiency at three different levels inside datacenters. The first work begins by invesitgating Facebook's production datacenters, and find that datacenter operators' oblivious choice of grouping servers serving the similar kinds of applications together creates rapid peaks of power consumptions at local power delivery devices. The power budgets of the local devices are consumed quickly during certain hours of a day, but are highly underutilized during other period of time. This inbalance use of power budget degrades the overall utilization of power budgets supplied by the power delivery infrastructure, leading to **datacenter-wise power inefficiency**.

The second work narrows our focus onto examining the efficiency of power management mechanism employed in individual servers. Commodity servers employs conventional coarse-grain dynamic frequency and voltage scaling (DVFS) based mechanisms and policies as their tool for power management. This coarse-grain mechanism, however, fails to identify long running queries and fails to selectively boost processor performance only for the urgent situations. With these failures, processors wastes significant amounts of power and energy boosting the already-fast-enough queries, leading to **server-level power inefficiency**.

The third work takes one step further, investigating and addressing the **inefficiency of the computing fabrics**. In datacenter environment, general-purpose computing fabrics such as CPU and GPGPU are commonly deployed because these computing units are designed to be capable of performing a wide spectrum of computation tasks. This flexibility, however, is achieved by generalizing all kinds of computation into some standard steps, each of which is performed by a pipeline stage in processor. In this situation, unfortunately, even the simplest operation needs to trigger most stages of the processor pipeline and potentially involves significant amounts of data movement in the memory hierarchy. Hardware accelerators are designed to address this type of efficiency issue by building specialized modules to accelerate cer-

tain computation tasks, eliminating the unnecessary execution and data movement. However, in datacenter environment, hardware accelerators suffer from its low flexibility and are difficult to be deployed to handle the fast changing datacenter workload. Our third work aims to achieve the best of both worlds, trying to realize dynamic and flexible computation on highly power efficient and reconfigurable hardware platform.

In the following subsections, we describes the motivation and the challenges of addressing the power inefficiency at each of these three levels in details.

1.1 Motivation and Challenges

1.1.1 Inefficient Datacenters

We investigate the power delivery infrastructure at Facebook production datacenters, and observe that the power provisioning in these datacenters faces two major challenges:

1. **Challenge 1: Peak provisioning leads to low power budget utilization.** For modern large-scale datacenters that mostly serve user-facing workloads, their total power consumption usually follows a *diurnal pattern* [102, 117], which reflects the highly fluctuating user activity level throughout a day. To ensure sustainability and safety, datacenter managers need to ensure that the peak aggregate power consumption can be accommodated under the given, fixed power budget supplied by power infrastructure. *Peak provisioning*, however, usually leads to highly underutilized power budgets during the rest of the day.

2. **Challenge 2: Multi-level power delivery infrastructure leads to power budget fragmentation.** Most modern datacenters deploy a multi-level power delivery infrastructure. These infrastructures have a tree-like structure, each node being a power delivery device, or we call a *power node*. Servers are directly supplied by lowest-level (*leaf*) power nodes, which are supplied by higher-level power nodes.

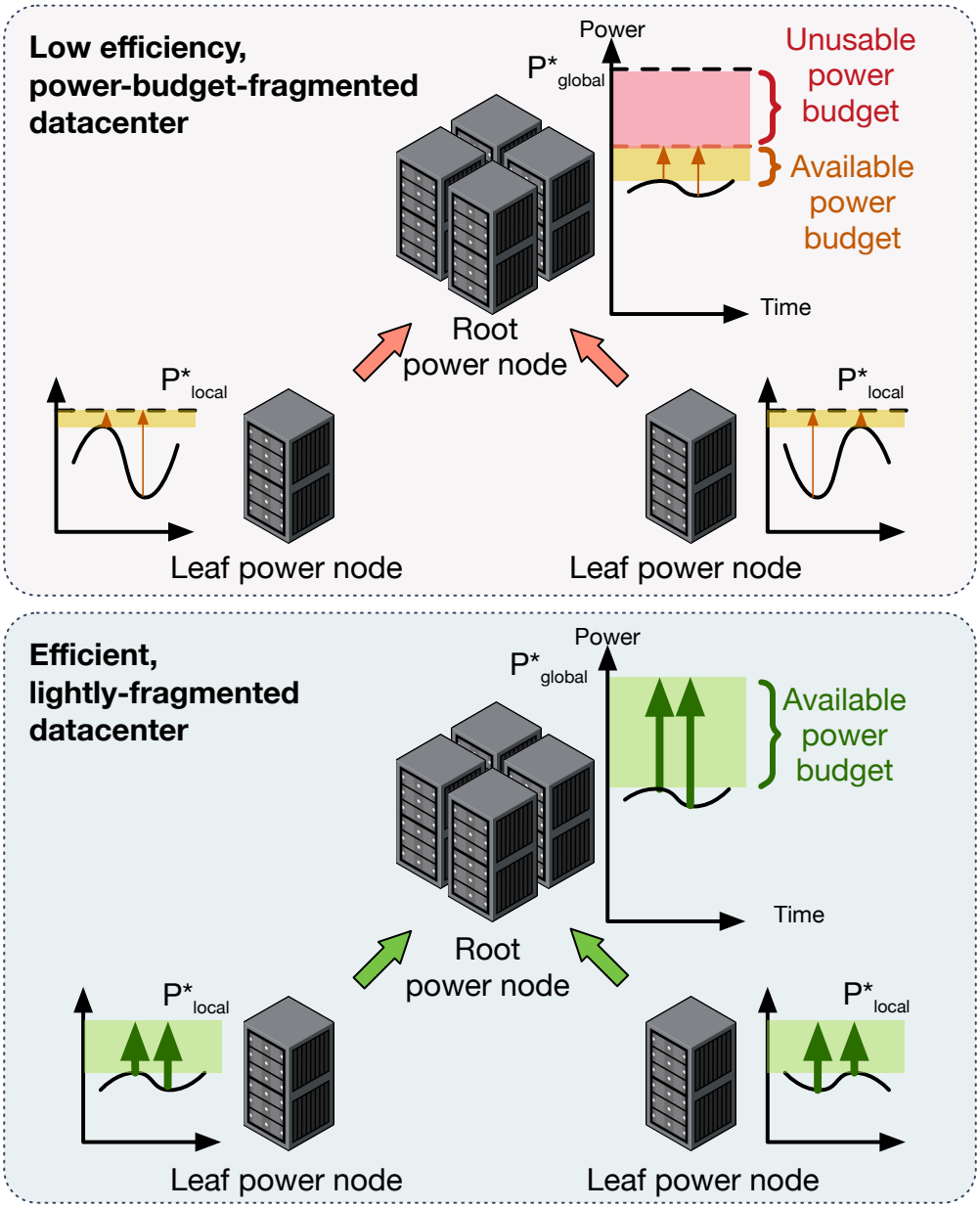


Figure 1.1: Grouping servers with synchronous power consumption patterns together often lead to rapid local peaks, which consume the local power budgets quickly and lead to significant amount of power budget being unusable.

Such hierarchical infrastructure improves reliability and is a common practice in large-scale datacenters. Unfortunately, it also leads to negative effects on power budget utilization because peak provisioning occurs at all levels of power nodes.

Power budget fragmentation problem, as illustrated in Figure 1.1, frequently occurs in multi-level power infrastructures. For instance, in Figure 1.1, we show two simplified datacenters, each of which is equipped with a two-level power infrastructure, and the same set of servers and service instances. The only difference between these two datacenters is how service instances are placed under the leaf power nodes. In the first datacenter, servers having synchronous power consumption patterns are connected to the same leaf node. This creates rapid peaks with high amplitudes at the leaf nodes, which consume the local power budget quickly. In such a datacenter, while there is still an abundant amount of power headroom at the root node, there is no room to connect any more servers to these leaf power nodes. *Since servers can only be supplied by the leaf power nodes, if the power budget is highly fragmented at the lower level of power delivery infrastructure, the abundant power headroom at the root node can never be utilized, making the datacenter inefficient.* In the second datacenter, on the contrary, servers are mixed in a manner that service instances with synchronous power consumption patterns are spread out. Our insight is *when carefully spreading out synchronous service instances, rapid peaks at leaf power nodes are eliminated, and power headrooms at the leaf nodes are increased.* These increased local power headrooms allow more servers to be supplied, which improves the utilization of the power budget and the overall datacenter efficiency.

A large body of prior solutions, including power capping [24, 44, 83, 123, 49, 31, 125, 57, 25, 16, 150, 148], were proposed to solve Challenge 1. When applying these prior works in a datacenter with oblivious service placement, however, their potentials are largely limited by power budget fragmentation. In such a datacenter, instances of the same services are typically placed together. Since these instances reach their peaks

around the same time, the corresponding leaf nodes that supply these latency-critical servers consume their power budgets much faster than the other leaf nodes. In this case, unfortunately, they need to be largely capped, even when there are still ample amounts of power headroom at other leaf nodes. A few techniques [78, 47, 115, 54] were proposed to address Challenge 2 in order to make power capping more efficient. These solutions, however, either require modifications to the power infrastructure [78, 115, 54], or due to the battery capacity can only handle peaks that span at most tens of minutes, making it unsuitable for Facebook type of workloads whose peak may last for hours. [47].

1.1.2 Inefficient Servers

Managing high-percentile tail latency is one of the chief performance concerns for web services in modern datacenters [32]. These online data-intensive (OLDI) services traverse multi-terabyte data sets for user-facing latency-sensitive queries by dividing ("*sharding*") their datasets over thousands of servers ("*leaf nodes*") acting in concert [12]. A complete query response is formed by aggregating the responses from the individual leaf nodes. However, at hundred- to thousand-node scale, the overall latency distribution to respond to the user is often dominated by a single straggling leaf node. For example, if an individual leaf node has only a 1-in-100 chance of exceeding a one-second response time, when aggregating parallel requests to 100 nodes, 63% of queries will take longer than one second [32]. Due to this service-level sensitivity to leaf-node tail latency, optimizations to reduce the long tail are paramount. Indeed, sacrificing mean latency for improved tail latency is encouraged [32].

Recently proposed workload- and latency-aware mechanisms, such as PEGASUS [90], adjust voltage/frequency to the large diurnal variations in service load to conserve energy while respecting a tail latency constraint. Such approaches similarly

fail to distinguish between typical and tail queries, instead modulating performance of the entire query latency distribution to meet the tail latency constraint. Coarse-grain uniform boosting leads to energy-inefficient tail reduction; energy is wasted accelerating queries that are not in the tail.

Considering that the query latency for many OLDI services is in the range of milliseconds and microseconds [88, 12], the emerging class of fine-grain (10s of nanoseconds) voltage boosting (i.e., *quick boosting*) techniques [72, 71, 119, 106, 53] has the potential to enable precise query-level boosting approaches. Given an energy budget, an intelligent quick boosting strategy could precisely pinpoint and boost queries that contribute to the tail as well as those whose latency is more likely to benefit from frequency/voltage boosting. Figure 1.2 illustrates the limitation of prior work and our insight. The top graph illustrates a typical heavy-tailed latency distribution of a leaf node. Coarse-grain boosting techniques such as those mentioned above accelerate all queries, compressing the entire latency distribution until the 99% latency meets the target, unnecessarily accelerating the bulk of requests near the mean. Instead, by applying fine-grain boosting to queries that are both sensitive to frequency and lie in the tail, our approach skews the distribution, significantly reducing the tail latency. However, several open questions remain to realize this approach, including:

1. **Investigating whether tail queries are amenable to frequency/voltage boosting.** If tail queries in representative OLDI workloads are not bottlenecked on computation, V/f boosting will not be effective in reducing their time to completion and thus would not be able to pull in the tail.

2. **Determining whether tail queries are predictable.** If the queries that push out the long tail share common characteristics, we can use these characteristics to develop per-query indicators to pinpoint these queries for boosting.

3. **Identifying an effective system design to pinpoint tail queries and pre-**

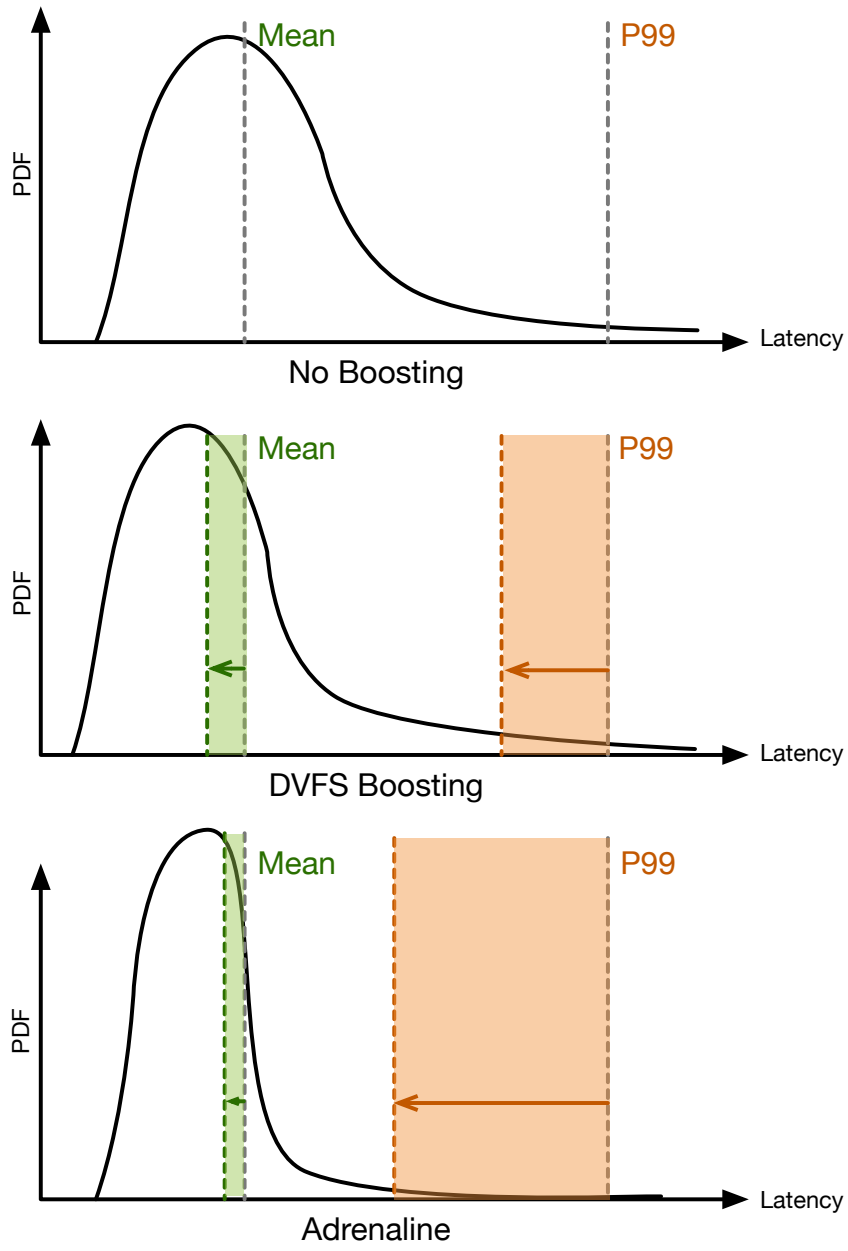


Figure 1.2: Query latency distributions without boosting (top), with conventional DVFS boosting (center), and with Adrenaline (bottom).

cisely boost them. To realize quick pinpointed boosting of tail queries, a mechanism must be in place to enable the identification and boosting of likely tail queries.

1.1.3 Inefficient Computing Fabrics

Despite declining CPU performance gains generation over generation, the computational requirements of cloud service datacenters are rapidly increasing. Current CPU-centric models will soon no longer be able to keep up with the demand placed on them by compute intensive, power-hungry workloads such as deep learning [58] and intelligent personal assistants [59].

Recent studies [59, 121, 21, 94] show that FPGA acceleration can close this gap between computing capacity and demand. FPGAs offer a number of benefits over other hardware accelerators: they boast significantly higher computational density for many emerging workloads and, unlike ASICs, can be reprogrammed to support alternate workloads and algorithms [88, 133, 20, 149].

Additionally, because the algorithm is explicitly implemented in hardware, FPGAs deliver extremely predictable performance [121, 21] (one of the most desirable characteristics in a cloud infrastructure [32, 136, 160, 80, 118, 61, 97, 134, 99, 26]). FPGAs are highly energy efficient and, unlike power- and space-hungry GPUs, FPGAs can be easily integrated into existing datacenter and server architectures [121].

There are several promising industry efforts to make FPGAs more accessible in the datacenter environment: Intel recently purchased Altera, (one of the two leading FPGA manufacturers) for $16.7 billion [73], and Xilinx (the other leader) is reportedly building close ties with Qualcomm [100] and IBM [63] to develop FPGA solutions for datacenters. Intel, following closely on the heels of their acquisition, released the first server prototype integrating a memory coherent FPGA with a Xeon processor over QPI. Called HARP, this unique platform is one of the systems we characterize in this

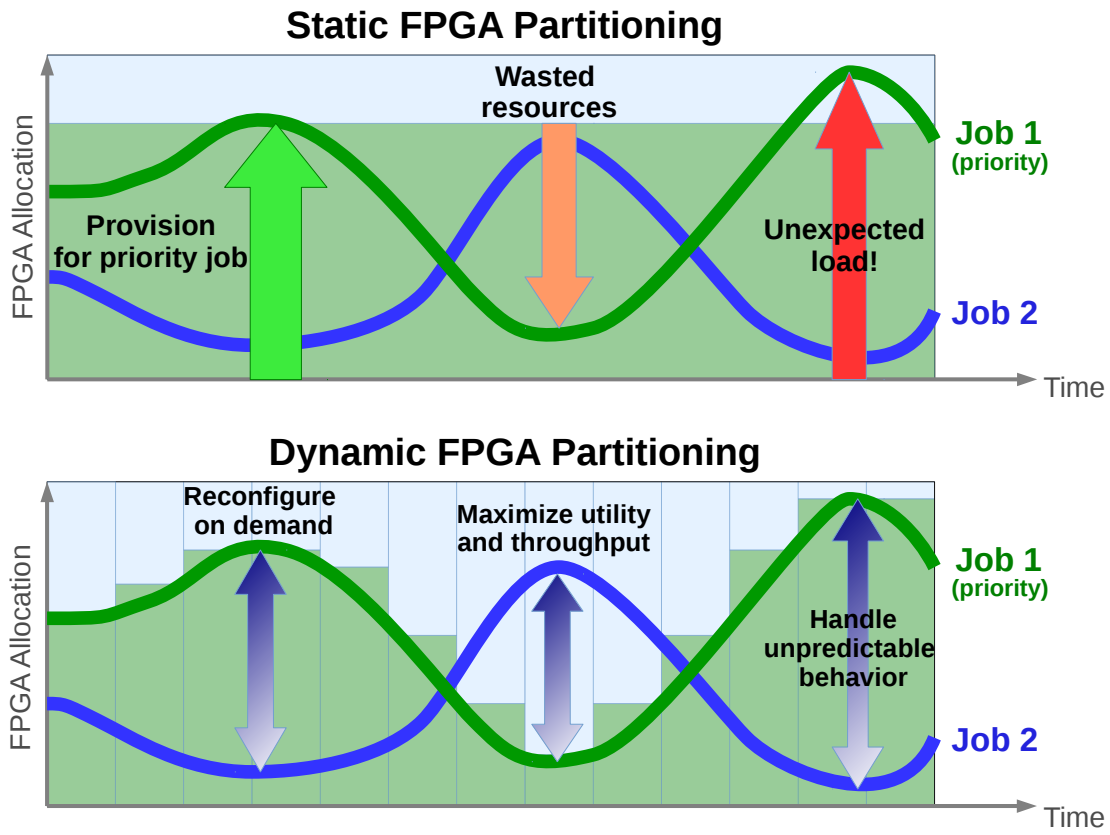


Figure 1.3: An example of FPGA fabric partitioning with multiple accelerators. Each GMM uses one tile, and AES and JPEG use hierarchical partitioning to use another tile.

work.

In this rapidly growing design space of FPGA-enabled datacenter computing, prior work statically assigns FPGAs to accelerate single workloads [121] or allocates entire FPGAs to be used for different workloads [21]. Statically allocating accelerators can leave FPGAs vastly underutilized, and allocating entire FPGAs for a workload can leave significant portions of the FPGA fabric unused while incurring a high reprogramming cost. This is both inefficient and expensive.

Modern datacenters must support many different types of jobs, and the demand for each type of job varies over multiple timescales (e.g., the diurnal utilization pattern reported by Google [103] as well as transient load spikes [142, 56]). As such, FPGAs need to be able to dynamically and quickly respond to both short-term and long-term

load changes across multiple workloads.

To solve this problem we create Slingshot, a resource management system that enables FPGA reconfiguration in response to real-time demand (Figure 1.3). Resource sharing is a particularly challenging problem because sharing the resources of an FPGA presents fundamentally different challenges from resource sharing on a chip multicore processor (CMP) with both spatial and temporal partitioning aspects:

- **Spatial Resource Sharing -** FPGAs offer a much finer degree of spatial resource sharing than CMPs, where the logical core is the smallest unit of spatial partitioning. An FPGA can be subdivided down to the individual logic components, and computing resources can be shared and allocated at a much finer degree.

- **Temporal Resource Sharing -** Modern CPUs can quickly switch between application contexts on the fly, however FPGA accelerators are limited to cooperative multitasking; that is, FPGAs can only reprogram to switch applications in between jobs. Additionally, the partial reprogramming time on an FPGA is over 100× higher than context switching time in a CPU [39, 85].

1.2 Summary of Proposed Systems and Contributions

This dissertation contains three major systems, each of which aims to address the power and energy inefficiency at a specific level. The summary of the systems and their contributions are stated as follows:

1. **Improving datacenter efficiency with SmoothOperator** – we aim to improve the efficiency of power usage in datacenters, allowing datacenters to achieve higher throughput without changing the existing power infrastructure. To this end, we propose **SmoothOperator**, a framework that analyzes the temporal heterogeneity of power consumption patterns and derives a highly power efficient service placement. SmoothOperator leverages a novel approach to systematically model

and scores the temporal heterogeneity among different services. Based on the analysis, SmoothOperator uses a clustering-based approach to identify instances that create high peaks when being placed together, and spreads them out across the datacenter.

This framework increases power headroom at all levels of the power delivery infrastructure, allowing hosting more servers and increasing the throughput of the datacenter. Meanwhile, we also observe that simply adding servers in a straightforward way can lead to resource underutilization and there are further opportunities to improve the throughput. We leverage a new type of disaggregate servers that decouple the compute and the storage components, recently deployed in production. We design a set of history-based server conversion and proactive throttling and boosting policies to fully utilize the newly-added servers to further increase the resource and power utilization.

2. **Addressing server inefficiency with Adrenaline** – We characterize query latency distributions for latency-sensitive datacenter applications running on real systems and identify *query-level indicators*. We then propose **Adrenaline** to address the limitations of traditional DVFS and achieve tail-sensitive voltage boosting to significantly reduce the tail latency with high energy efficiency. Adrenaline leverages the recently proposed *Shortstop* [119], a circuit design for quick voltage/frequency scaling, and the knowledge of per-query characteristics to achieve query-level fine-grain V/f scaling. Adrenaline's fine-grain query-level V/f boosting pinpoints the long-running queries and reduces the tail latency when needed, while still be able to significantly improve energy efficiency by only lowering V/f for queries that are less likely to increase the tail or less affected by the V/f scaling.

3. **Workload scheduling on low-power and dynamically configurable fabric** – Hardware acceleration provides both high power efficiency and highly pre-

dictable latency. We characterize the state-of-the-art dynamically-configurable CPU-FPGA systems with a set of representative computation kernels. We present the design of Slingshot, a robust mechanism to share FPGA resources spatially and temporally across multiple dynamic workloads. Using Slingshot, queries from different applications can be flexibly served throughout the datacenter without being restricted by the present configuration of FPGA resources, and each machine dynamically decides whether to process queries on the CPU or FPGA using resource-aware scheduling algorithms. The end result is that Slingshot effectively treats the FPGA as a shared service.

Using Slingshot's FPGA-as-a-service model, each server dynamically reprograms its own FPGA on the fly to best cope with changing load patterns (e.g., increasing the available accelerators of a particular type in response to a sudden spike in relative load). In this work, we construct a datacenter model based on the real-system measurements and study the impact of different partitioning and scheduling policies for dynamically reconfigurable FPGA acceleration.

CHAPTER II

Background and Related Work

2.1 Improving Power Budget Utilization for Clusters and Datacenters

Aside from prior proposals focusing on improving node-level power and energy efficiency [90, 91, 61], there has been a large body of previous works addressing power and energy efficiency problems in datacenters from a system architecture's perspective.

One class of solutions tries to address the power bottleneck by reducing the power consumption of computation. Power management problems at the server level and the small cluster level have drawn significant attention [43, 24, 45, 110, 123, 141, 87, 65, 131, 50, 124, 103, 35, 140]. A popular approach among them relies on using virtual machines (VMs) to achieve work consolidation and performance isolation among different workloads, and treats VM consolidation as a resource allocation problem [24, 45, 110, 123, 141, 87]. However, due to strict latency requirement and simplicity of management, large-scale production datacenters, such as Facebook's and Microsoft's [161], deploy their datacenter without virtualization in a more autonomous manner. In this type of datacenters, each service team deploys their service on their own, separate set of physical servers, and different major services do not share phys-

ical servers. This consolidation-based approach is not directly applicable to this type of datacenters.

Recently, researchers started to consider intelligently charging and discharging energy storage devices (ESDs) to enable temporary excessive power draw at power nodes during power emergencies [6, 55, 155, 145, 78, 56]. The problem of this type of approach is similar to power-capping-based approaches, in that the unbalanced peaks across the datacenter can make the sharp peaks at some of the nodes deplete the ESDs quickly, while at some power nodes the power draw never entails the extra capacity.

There are other proposals for flexible power infrastructure focusing on temporarily extending (or reallocating) the power budget among power nodes [115, 41] to adapt to the dynamism of power consumption. Among them, power routing [115] suggests dynamically connecting rows and racks of servers to different power nodes, in order to balance the load among power nodes. Dual-corded power supply, however, only provides limited flexibility (degree of 2) for power routing purpose. To maximize the flexibility, normal fault-tolerant dual-corded power supply architecture must be expanded to enable richer connectivity. The costly upgrade to the infrastructure and a significant change to the power supply topology required by their solution can further lead to long service downtime during the installation and setup process.

Several prior works [44, 54, 144] showed that by statistically multiplexing the probability distributions (PDFs) of the power of different workloads, datacenters can overbook and/or under-provision safely. These works, however, do not take advantage of time-series power information of workloads. Our insight is that the strong diurnal patterns and the asynchronous power behavior across workloads provide significant opportunities for de-fragmenting power budget. Also, their techniques degrade the performance of user-facing services significantly during the peak time, which is not ideal for a user-facing datacenter.

	Power Routing [115]	Stat. Multiplexing [54]	DistributedUPS [78]	SmoothOperator
Using temporal information			✓	✓
Using existing power infra.		✓		✓
Automated process	✓	✓	✓	✓
Balancing local peaks	✓			✓
Proactive planning				✓

Table 2.1: Comparison between SmoothOperator and prior approaches for improving datacenter power efficiency

In fact, SmoothOperator and a lot of the works mentioned in this section are not mutually exclusive. The workload-aware service instance placement framework is complementary to many of the prior solutions.

2.2 DVFS and Server-level Power Management

Dynamic voltage and frequency scaling (DVFS) has been widely studied to improve the energy efficiency of various scales of computational units over the past several decades [27, 151, 64, 69, 82, 77, 92, 90]. However, most of the prior works only look at voltage and frequency scaling decisions at a coarse granularity. In [27, 151, 64], a decision engine, which leverages measured runtime information, is implemented as regression models or dynamic compilation mechanisms to find the best voltage and frequency levels to optimize for energy efficiency. Researchers have also put efforts into coarse-grain DVFS decisions on modern multi-core processors [64, 82, 77, 92], in order to achieve better system throughput and energy efficiency. Similarly to DVFS on processors, some prior works also explore the opportunities of deploying DVFS on memory systems [35, 30, 37, 36]. Related works on quick voltage boosting techniques are discussed in Chapter IV.

Recently, there is an increasing research interest on power management in datacenters [122, 84, 92, 90, 101] from different perspectives. Among them, PEGASUS [90] constantly monitors the workload and the performance statistics of the recent requests within a sliding window, and leverages a feedback controller to minimize the

power consumption without violating the QoS requirement. This work differs from theirs by identifying and selectively targeting only the requests that most likely will fall in the tail of the distribution.

2.2.1 Tail Latency

As reported in prior work [32], tail latency has become a major concern of modern datacenter applications, which has gained much research attention. Since many datacenter workloads have critical tail latency requirements, many works [80, 160, 154, 137, 98, 135, 96, 33] try to improve the performance predictability of such workloads to optimize datacenter utilization. This requires precise performance interference prediction and careful control of resource sharing, in order to make better scheduling decisions. Another class of optimization tries to reduce the tail latency. DeTail [157] proposes to exchange package information across multiple network layers to optimize the scheduling of package processing and distribute the network load evenly. [158, 13] move the network stack from kernel space to user space to avoid overhead, so that they can achieve lower query latency, as well as higher system throughput. Adrenaline differs from these works due to the fact that it specifically accelerates the queries that tend to appear in the tail, which makes the optimization more efficient than simply boosting the entire latency distribution.

2.3 FPGAs and Hardware Acceleration

As Moore's law slows down, new techniques are required to meet the growing computational needs of datacenter services [59]. Hardware accelerators can reduce computation time and increase efficiency for many types of algorithms including text and image processing [8, 120, 22, 66, 156], scientific computing [86, 7, 60, 113, 19], database transactions [10, 132, 108] and machine learning [112, 162, 79, 107, 5, 159].

FPGAs, ASICs, and GPUs each have their own strengths and weaknesses. GPUs

offer massive parallel computing capabilities, but use a lot of power [114, 67]. ASICs are very power efficient but expensive to implement and cannot be changed once built. FPGAs can be easily modified and are particularly good at accelerating algorithms that are amenable to a pipelined architecture, have frequent branches, and/or can benefit from hardware-level architectural optimizations [59, 70, 109].

Prior work demonstrates FPGAs can accelerate numerous workloads seen in datacenters today. Yan et al. showed that FPGAs can be used to accelerate query processing for web search engines by a factor of 7 compared to CPU [153] and Shan et al. demonstrated a greater than 30x speedup accelerating MapReduce on an FPGA [129]. However, the integration of accelerators into a datacenter has many factors besides implementing an algorithm: datacenter designers must account for networking latency, bandwidth limitations, and power constraints. In order for a datacenter to scale under load, all of these issues must be managed while scheduling jobs to maximize Quality-of-Service (QoS), throughput, and energy efficiency [121, 139, 11, 14, 18, 89, 68].

Prior work on multiple FPGAs in a cluster or datacenter focuses on one of two options: using all the FPGAs for a single task (platform-as-a-service model) or scheduling on bare-metal (infrastructure-as-a-service model).Sano et al. used the platform-as-a-service model to show that FPGAs can be pipelined together to complete a single workload [128]. Putnam et al. demonstrated a medium-scale deployment of Stratix V FPGAs can be used to accelerate the Bing web search engine [121]. Both use a reconfigurable fabric but statically partition the entire fabric for a single workload, unlike our proposed system which dynamically partitions the reconfigurable fabric in response to load needs.

The infrastructure as a service model is a special case of heterogeneous datacenter scheduling, with very diverse hardware choices. Lee et al. showed that for heterogeneous cloud scheduling, a reasonable scheduling goal is the cost-performance tradeoff [81]. The cost determines the metrics of hardware cost and execution time,

and the scheduler must track the utilization of each compute resource to optimize for performance. We aim to be the first work to utilize and characterize the hardware heterogeneity of Intel's HARP for dynamic datacenter workloads, where the FPGA has a low latency pathway to memory.

For scheduling on hardware, El-Araby et al. proposed a virtualization model for FPGA-accelerated systems where an FPGA is partitioned and IO pipelined to different regions [40]. Similar to virtual memory, this allows an application to *see* more resources than are currently available and delegates the actual reprogramming and data transfer work to a scheduling algorithm. However, this does not account for workloads with different computation-to-data ratios, reprogramming latency, or contention and latency from other sources in a datacenter that uses to make query scheduling and FPGA reconfiguration decisions.

For contention-aware workload scheduling in a datacenter, Paragon, a QoS-Aware scheduler for heterogeneous datacenters, characterized workloads and hardware resources and uses a greedy algorithm to try to match jobs with available resources [34]. Paragon applicability to an FPGA-accelerated datacenter is limited by the fact that it cannot characterize accelerated workloads. Dominant Resource Fairness (DRF) [51] showed that an efficient, fair, and Pareto-optimal scheduling algorithm that accounts for and distributes all the resources needed by each workload, but cannot manage the reconfiguration of FPGA resources and does not provide sufficient support for dynamically changing resource configurations.

CHAPTER III

SmoothOperator: Reducing Power Fragmentation and Improving Power Utilization in Large-scale Datacenters

This chapter presents **SmoothOperator**, a datacenter-level framework designed to address the power and energy efficiency in production datacenters. We develop a *workload-aware service placement* framework to systematically spread the service instances with synchronous power patterns evenly under the power supply tree, greatly reducing the peak power draw at power nodes. We then leverage *dynamic power profile reshaping* to maximally utilize the headroom unlocked by our placement framework.

3.1 Power budget fragmentation and inefficiency

3.1.1 Multi-level power infrastructure in datacenters

Multi-level power infrastructure is commonly deployed in large-scale, production datacenters. At each level, the total power budget of a power node is shared among its children nodes. This type of tree-like multi-level power infrastructure is designed to avoid a single point of failure and for the convenience of management.

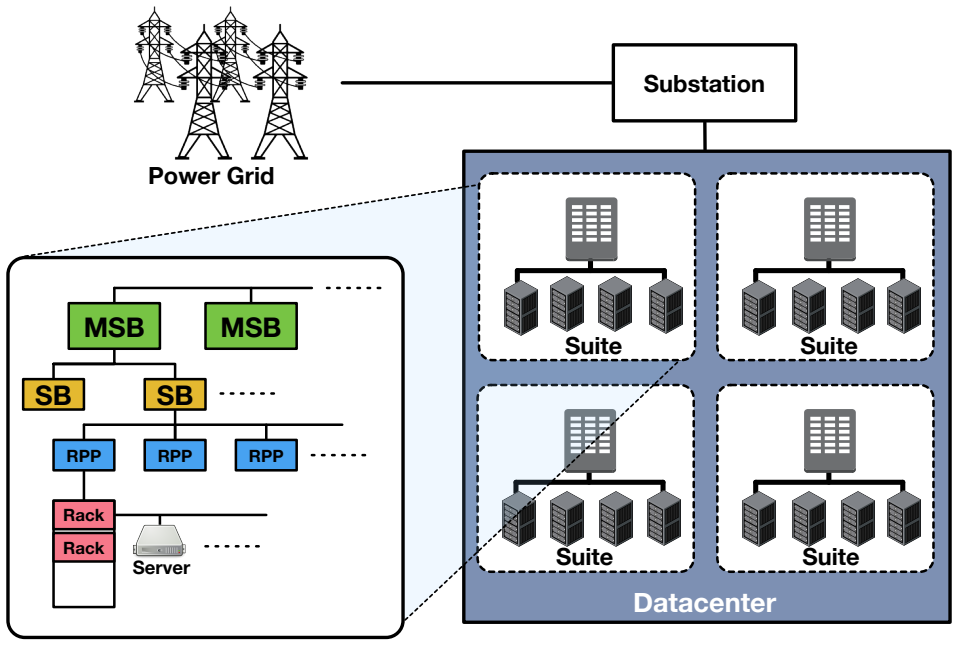


Figure 3.1: Multi-level power delivery infrastructure deployed in Facebook's datacenter.

Facebook datacenters feature four-level power infrastructure, consistent with Open Compute Project specification [4, 150], as shown in Figure 3.1. Each datacenter is composed of several rooms, which we call *suites*. Servers are placed onto rows of racks, allocated into these suites. A suite is equipped with multiple top-level power nodes, i.e., main switching boards (MSBs), each of which supplies some second-level power nodes, switching boards (SBs), which further feeds to a set of reactive power panels (RPPs). Finally, the power is fed into racks, each composed of tens of servers. The power budget of each node is approximately the sum of the budgets of its children.

3.1.2 Power budget fragmentation

Power budget fragmentation exists because servers hosting the services with synchronous power consumption patterns are grouped together under the same sub-tree of the multi-level power infrastructure. Such a placement creates rapid, unbalanced peaks with a high amplitude, which consumes the power budget of the supplying

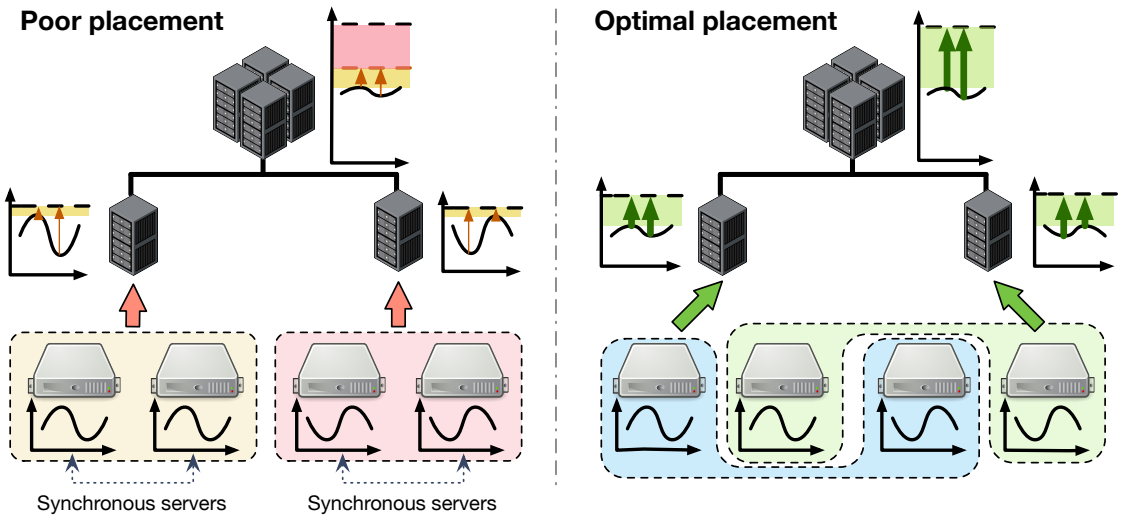


Figure 3.2: Careful service instance placement can extract significant power headroom for housing more servers, which improves the overall computing capacity of the datacenter without changing the underlying power infrastructure.

power node fast. When the aggregate power at a power node exceeds the power budget of that node, after a short amount of time, the circuit breaker is tripped and the power supply for the entire sub-tree is shut down. These local power limits, therefore, make it harder to efficiently utilize the total power budget that is supplied to the entire datacenter.

For example, in Figure 3.2, we have a simplified datacenter with a 2-level power infrastructure and 4 service instances to be placed under the leaf power nodes. We assume that service instance 1 and 2 have an identical (perfectly *synchronous*) power consumption pattern, and service instance 3 and 4 have perfectly out-of-phase patterns. To avoid tripping the circuit, the limiting factor of the number of servers that can be supplied under a power node is the peak power (maximum aggregated power across all servers supplied by the same node). When synchronous servers are placed together, as shown in the left sub-figure of Figure 3.2, it leads to a higher peak power at the supplying power node than the optimal placement. This higher peak curtails the number of servers that can be supplied by the datacenter, which indicates a lower

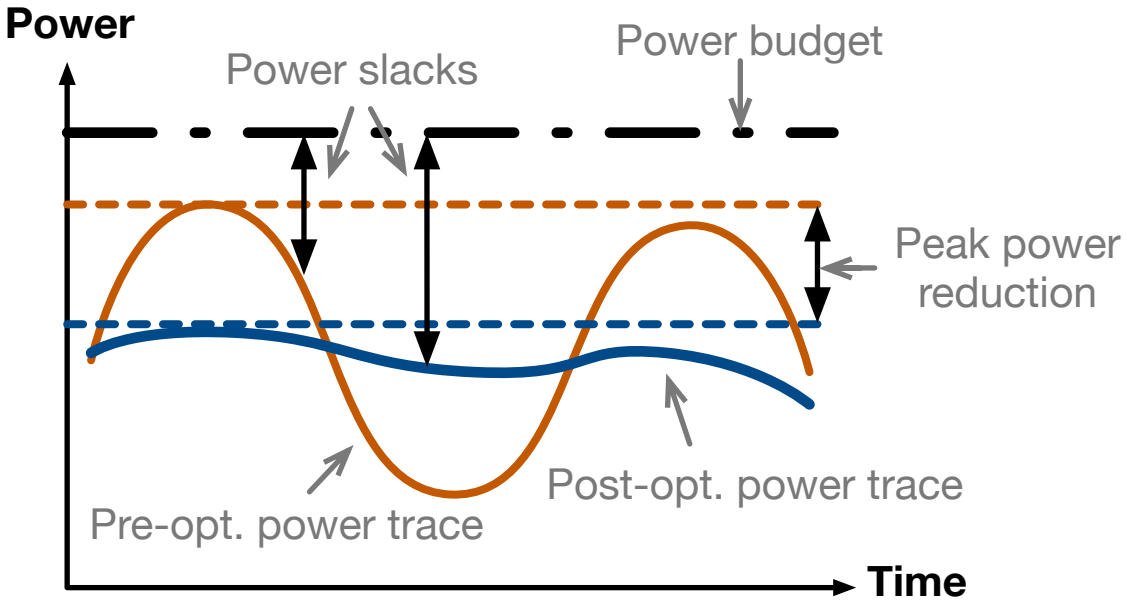


Figure 3.3: Power Slack, defined as the difference between the current power consumption at time t and the power budget. It quantified the power utilization efficiency.

power utilization.

In the following, we define two metrics to quantify the level of power budget fragmentation and inefficiency of power budget utilization. In this work, we focus on improving these two metrics:

1. **Sum of peaks:** Sum of the peaks of the power nodes in a datacenter is an important indicator of the level of power budget fragmentation. With the same set of service instances, poor placements can produce very a high peak power value at leaf nodes, indicating the peaks of the service instances are not evenly spread out. The sum of the power node peaks is, therefore, much larger than that in the optimal placement.

2. **Power slack and energy slack:** Power slack and energy slacks are indicators for power budget utilization. As illustrated in Figure 3.3, *power slack* is a mea-

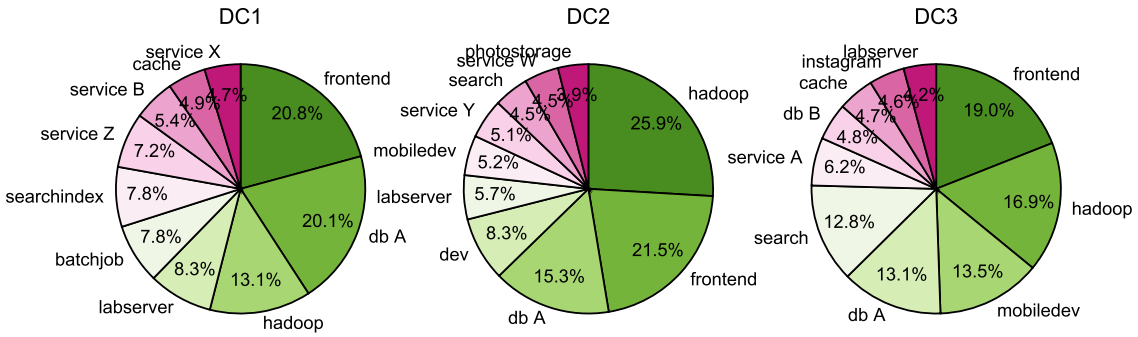


Figure 3.4: The breakdown of the 30-day average power consumption of the top 10 power consumer workloads measured in three of the datacenters under study.

surement of the unused power budget at a point of time, and is defined as

$$P_{slack,t} = P_{budget} - P_{instant,t}, \tag{3.1}$$

where $P_{instant,t}$ is the instant power consumption of the power node at time t, and P_{budget} is a constant number representing the given power budget of this power node. The lower the power slack, the higher proportion of the power budget is utilized at that point of time. *Energy slack* is simply the integral of power slack over a timespan T.

$$E_{slack,T} = \int_T P_{slack,t} dt \tag{3.2}$$

A low energy slack means the power budget is highly utilized over the corresponding timespan.

In the following sections, we will use these two metrics to guide the development of our solutions and measure the quality of our result.

3.1.3 Peak heterogeneity in datacenters

Modern datacenters often provide a wide spectrum of services. Even in highly web-centric companies such as Facebook, to support a variety of data-centric features and

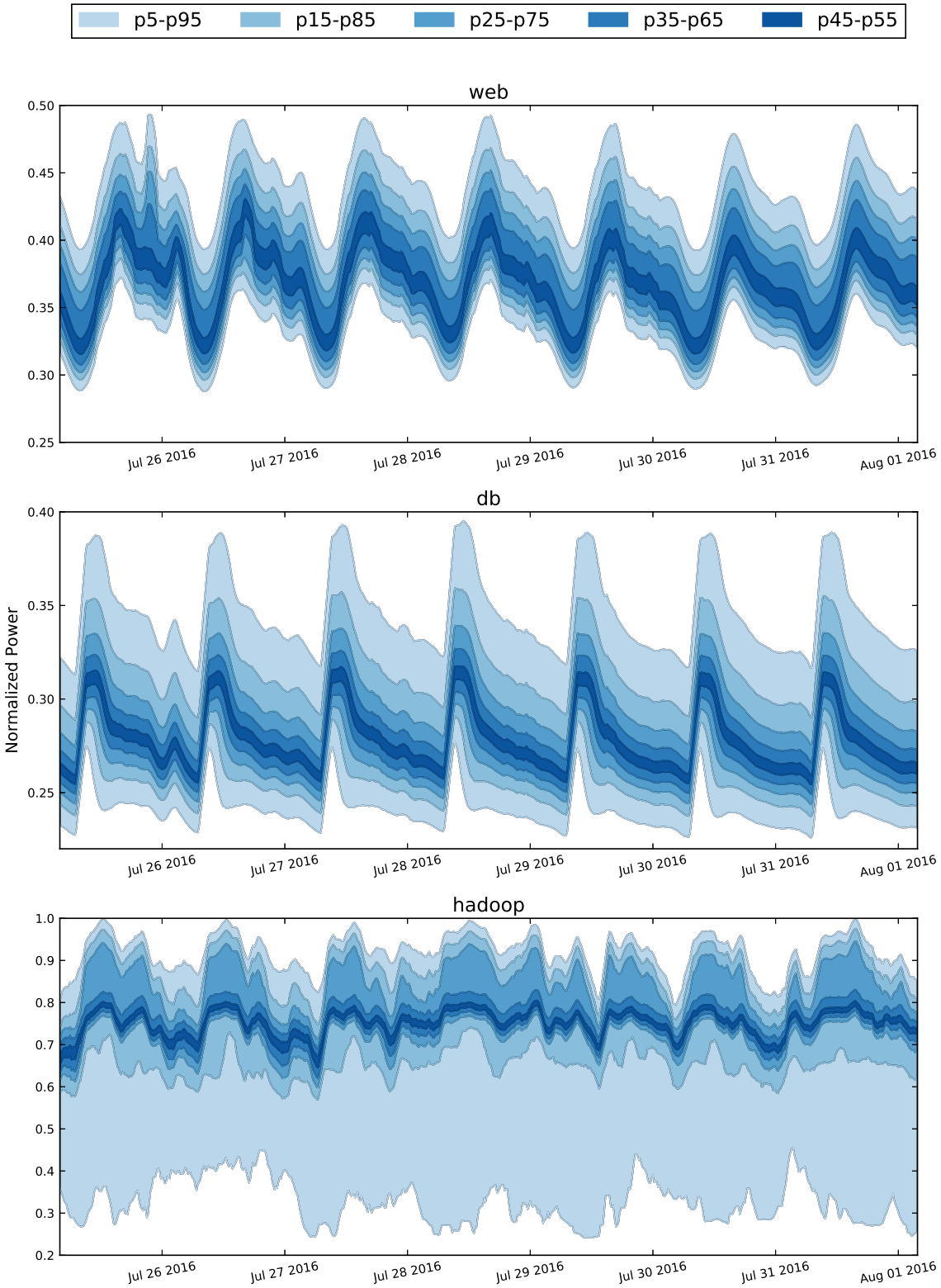


Figure 3.5: Servers serving different types of workload has very distinctive diurnal patterns. Servers included in this distribution is sampled from one of DC1.

the high visit rate, a significant proportion of servers in their production datacenters are provisioned to serve hundreds to thousands of internal services. Figure 3.4 presents the breakdown of 30-day average power consumption of the top 10 power consumer workloads measured in the three datacenters under study.

We observe that the power usage patterns are heterogeneous across these services. Such heterogeneity indicates abundant opportunities to mitigate the power budget fragmentation problem by grouping services with complementary power consumption patterns under the same power node. Figure 3.5 shows the diurnal patterns of three of the major services that are hosted in one of Facebook datacenters: **web**, **db**, and **hadoop**. The bands indicate the percentiles of the power reading among all the servers hosting that service. For example, the darkest band in the top-most sub-figure indicates the range between the 45th-percentile and the 55th-percentile power consumption readings among all the web servers at any given time. From this figure we show that servers hosting different services have very different power consumption patterns.

The web clusters serve the traffic coming directly from the users and hitting the production web site. This type of clusters, including web and cache servers, is the major part of Facebook datacenters, and is one of the largest consumers of the power budget. Because of its user-facing nature, the servers in these clusters have highly synchronous power patterns, which follow the typical user activity level. Meanwhile the latency requirement for this type of workload is high because it directly affects user experience.

The db clusters are the backend of the web and cache clusters. What distinguishes backend servers from these front-end servers is that query only hits db servers when the data needed are missing in the cache servers. Compared to front-end servers, these servers are more I/O bound, thus not exhibiting high power consumption even when the frontend servers are experiencing peak usage during daytime. However,

these servers perform daily backup at night, which involves a lot of data compression. Therefore, as shown in Figure 3.5, while these servers also have predictable diurnal pattern, their peaks occur during the night time.

The hadoop clusters serve batch jobs that aim to improve the quality of data that are needed by the website or the company. Since users don't directly interact with this type of servers, these services are not bound by latency requirement; instead, they are optimized to provide high throughput. To achieve satisfactory throughput, the execution of this type of jobs highly relies on the job scheduler's policy, and the server are running at higher settings of CPU frequencies. Consequently, we find in Figure 3.5 that their power consumptions are constantly high and less relevant to the user activity level.

We see from the above that there are abundant opportunities to improve the datacenter power budget utilization if the datacenter operators take advantage of different characteristics of different workloads. Recognizing such opportunities, we design SmoothOperator to capture and leverage the peak heterogeneity identified among these production services, in order to mitigate the power budget fragmentation problem by intelligently grouping the servers with asynchronous power consumption patterns. We will introduce how we design SmoothOperator in Sections 3.2 and 3.3.

3.2 Workload-aware service instance placement and remapping

In this section, we address the power budget fragmentation problem found in production datacenters, and introduce our *workload-aware service instance placement and remapping* framework. Our framework takes advantage of the service-level and service-instance-level heterogeneity, and spreads out service instances with synchronous power patterns under different power nodes.

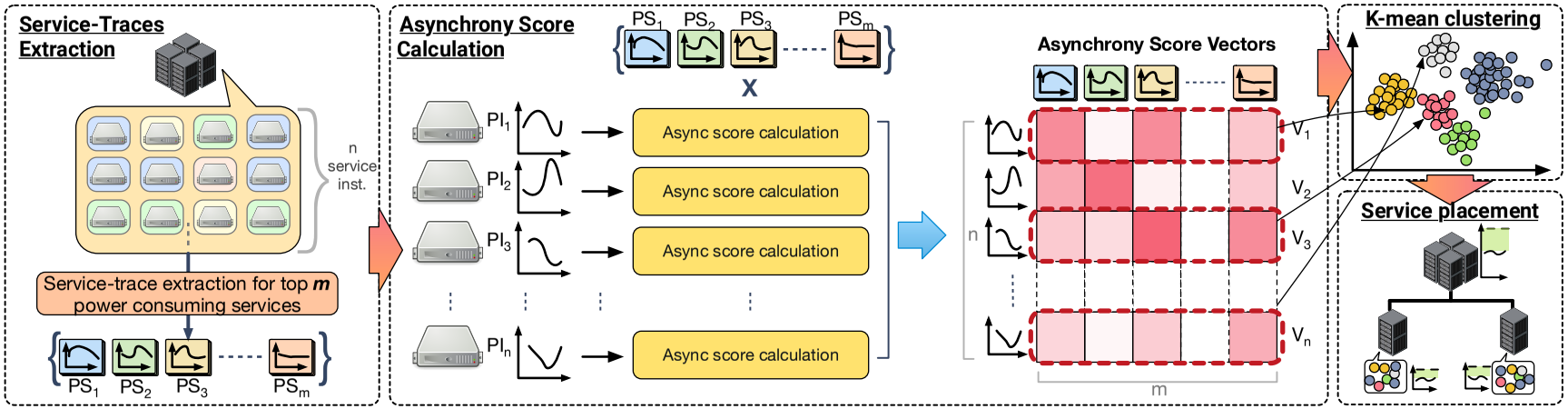


Figure 3.6: The overview of workload-aware service placement.

3.2.1 Service and service instance

Facebook datacenters house thousands of web and data processing services. Similar to Microsoft datacenters[161], for major services, each service team manages its own *service* on a separate set of physical servers, and different major services do not share physical servers. A service is a collection of hundreds to thousands of *service instances*. Each of these service instances is a process that runs a copy of the service or a part of the service. For example, service such as Memcached runs on thousands of machines, and each Memcached process on a server is a service instance. Facebook deploys service instances as native processes instead of virtual machines. This policy not only reduces operational complexity, but also minimizes the variability caused by the interference due to co-scheduling and colocation.

3.2.2 Overview of placement framework

We illustrate our framework in Figure 3.6, which includes four major steps:

1. **Collect traces and extract representative traces** We first collect and construct multi-weeks of power traces for each service instance, which we call *instance power trace*, of the target datacenters ($\boldsymbol{PI}_i$ in Figure 3.6). We use these instance-level traces to further construct a *service power trace* ($\boldsymbol{PS}_i$) for each of the top power-consuming services. Each *service power trace* exhibits the representative temporal patterns of a service aggregated across all its instances. The service-level traces then serve as a set of bases that facilitates the evaluation of dissimilarity between instance-level power traces (Section 3.2.3).

2. **Calculating asynchrony scores** We then identify synchronous service instances, namely, the instances whose power consumption peak at the same time. To achieve that, we calculate a vector of *asynchrony scores* for each service instance based on the corresponding instance power trace and the service

power traces of all the services (Section 3.2.4). We use the vectors of asynchrony scores to estimate the impact to the aggregated peak when two or more service instances are grouped together. This step transforms each server into a data point in a high dimensional space spanned by the asynchrony scores.

3. **Clustering** We then apply a clustering method (Section 3.2.5) on these service instances based on their asynchrony score vectors, identifying the ones with synchronous power consumption behavior.

4. **Placement** We place the service instances based on the clustering result, aiming to maximize the asynchrony score of each power node (Section 3.2.5).

After the initial application, our framework can be continuously applied to the datacenter to fine-tune the placement when power consumption patterns start to exhibit middle-term or long-term (e.g., in weeks or longer) shifts or changes.

Note that workload-aware service instance placement also provides benefits from the power safety aspect of datacenter. In the optimized placement, service instances that have highly synchronous behaviors are now spread out evenly across all the power nodes. When bursty traffic arrives, the sudden load change is now *shared* among all the power nodes. Such load sharing leads to a lower probability of high peaks aggregates at a small subset of power nodes, and therefore decreases the likelihood of tripping the circuit breakers inside certain heavily-loaded power nodes.

3.2.3 Constructing power traces

To be able to derive a workload-aware instance placement for datacenters, we need to capture the power consumption patterns of service and service instances. As shown in Section 3.1.3, services exhibit a diversity of power consumption patterns [161, 48] and such service-level power heterogeneity provides abundant opportunities for mitigating the fragmentation problem. In fact, in addition to service-level

heterogeneity, we observe from the production data significant amounts of *instance-level heterogeneity*, even within the same service. Such heterogeneity usually stems from imbalanced accessing pattern or skewed popularity among different instances of a same service. SmoothOperator aims to capture both of these two types of heterogeneity by constructing *instance power traces* and *service power traces*. In the following, we introduce the details of these two types of power traces.

Constructing Instance Power Traces For every service instance in the datacenter, we construct a log of power readings, i.e., an *instance power trace (I-trace)*, to represent the corresponding service instance's history of power consumption. Each of these I-traces is a *time series*, which is a vector, containing seven days of the exact power reading recorded by the power sensor on the corresponding machine, one reading per minute.

$$\boldsymbol{PI}_{i,w} = \langle p_{i,t} : t \in T_w \rangle, \tag{3.3}$$

where $PI_{i,w}$ is the instance power trace of server instance i of week w, and $p_{i,t}$ is the power reading of instance i at time t, and T_w is the series of timestamps within week w during which the power readings are logged. We choose the length of 7 days because, in large-scale user-facing datacenters, user traffic has strong day-of-the-week activity patterns [15, 126]. Note that, since power traces are simply vectors, vector arithmetic can be directly applied.

SmoothOperator focuses on balancing the power load of the power nodes by investigating the service instances' middle-term (e.g., hourly, daily) to long-term (e.g., weekly or longer period) power consumption patterns. To prevent SmoothOperator from overfitting its decisions to any specific week in which significant unusual short-term variations exist (e.g., bursty traffics due to power failure of neighboring datacenters), we collect 2-3 weeks of I-traces for each service instance, and use vector arithmetic to calculate the *averaged instance power trace* for each service instance.

That is,

$$\bar{\boldsymbol{PI}}_i = \frac{\sum_{w \in W} \boldsymbol{PI}_{i,w}}{|W|} \tag{3.4}$$

Each of these averaged I-traces remains to be a 7-day-long vector; each element of this averaged instance trace is the average of the power reading recorded at the same time-of-week across these multiple weeks.

Constructing Service Power Traces. We then construct *service power traces (S-traces)*, one for each of the top power-consumer services running in the datacenters. For service Y, we calculate the vector sum of the $\bar{PI}$'s of all of Y's instances, and divide the vector sum by the number of instances of service Y. This calculation is formulated as follows:

$$\bar{\boldsymbol{PS}}_Y = \frac{\sum_{service(i)=Y} \bar{\boldsymbol{PI}}_i}{|Y|}, \tag{3.5}$$

where $\bar{\boldsymbol{PS}}_Y$ is the service power trace of service Y, and $|Y|$ is the number of instances of service Y. These S-traces represent the most significant power consumption patterns observed in the datacenters; it means that, when randomly sampling any large enough group of service instances from the datacenter, the aggregate power trace of this group of service instances will be close to a linear combination of these top-consumer S-traces. When considering adding an extra service instance to a group of instances, we use these S-traces to evaluate whether the new instance's power consumption pattern will add significantly to the peak of the aggregate power trace of that group.

3.2.4 Asynchrony score function

To measure how the peaks of the power traces of a set of service instances spread out over time, we define a metric, namely *asynchrony score*. We use an *asynchrony score function* to evaluate the asynchrony score over a set of power traces M, which

is defined as follows:

$$A_M = f(M) = \frac{\sum_{j \in M} peak(\bar{\boldsymbol{P}}_j)}{peak(\sum_{j \in M} \bar{\boldsymbol{P}}_j)}. \tag{3.6}$$

For example, if we want to evaluate whether two service instances a and b should be placed together, we would like to know if they peak asynchronously or not. Since an I-trace is a power trace, we can calculate asynchrony score function over two I-traces. This is done by calculating the following ratio for $\bar{\boldsymbol{PI}}_a$ and $\bar{\boldsymbol{PI}}_b$, and the aggregate power trace $\bar{\boldsymbol{P}}_{\{a,b\}} = \bar{\boldsymbol{PI}}_a + \bar{\boldsymbol{PI}}_b$:

$$A_{\{a,b\}} = f(\{a,b\}) = \frac{peak(\bar{\boldsymbol{PI}}_a) + peak(\bar{\boldsymbol{PI}}_b)}{peak(\bar{\boldsymbol{P}}_{\{a,b\}})}, \tag{3.7}$$

where $A_{\{a,b\}}$ is the *asynchrony score between service instance a and service instance b*, and $peak(\bar{\boldsymbol{P}}_j)$ is the peak value of the power trace $\bar{\boldsymbol{P}}_j$. The lower the asynchrony score, the more overlapping the peaks of the component power traces, and therefore the *worse* the group is; the higher the score, the less the overlap. For example, in the poor placement case in Figure 3.2, each leaf node has a asynchrony score of 1.0. If we exchange server 2 and server 3, each of the leaf power nodes will have a asynchrony score close to 2.0.

For a set of power traces M, the lowest possible A_M is 1.0, meaning that every component power traces peaks at the same time. The highest asynchrony score, $|M|$, occurs when every instance has the same peak value p and the peak of the aggregate power trace of M is also p, meaning that the aggregation of this group of instances has zero impact on the peak. This scenario represents the most efficient use of the power budget.

3.2.5 Service instance placement

Our workload-aware service instance placement mechanism relies on iterative calculation of asynchrony scores to derive a placement of service instances that leads

to more stable, less varying aggregated power consumption under power nodes. The resulting placement has a high asynchrony scores at all levels of power nodes, which maximizes the power headroom and mitigates fragmentation in the datacenter, as we described in Section 3.1.2. In the following, we dive into the details of our service instance placement mechanism.

Calculating asynchrony score vectors for service instances. We first calculate a *asynchrony score vector* v_i for each service instance i. We do this by calculating the asynchrony score of the averaged I-trace of instance i against all the S-traces; each element of this asynchrony score vector which is an asynchrony score between the averaged I-trace of i and one of the S-traces, which we call an *instance-to-service (I-to-S) asynchrony score*. This vector evaluates how adding this service instance to a large mix of other instances potentially impacts the aggregate peak of this group.

We choose to use I-to-S asynchrony scores, instead of instance-to-instance (I-to-I) asynchrony scores, for the evaluation because of two reasons: 1) the pair-wise I-to-I asynchrony score calculation could take an unacceptable amount of time since there are tens to hundreds of thousands of service instances in a datacenter, and 2) after doing the embedding in the next step, these I-to-I asynchrony score vectors will span a sparse high-dimensional space ($> 10^4$), which can lead to overfitting and is not ideal for clustering [130].

Classifying and placing instances. Suppose B is the set of the top power-consumer services, we extract $|B|$ S-traces. Each service instance, hence, has an asynchrony scores vector of length $|B|$. We can then embed all the service instances as data points in a $|B|$-dimensional space spanned by the $|B|$ asynchrony scores. We then apply *k-means clustering* to these points to classify the service instances into highly-synchronous groups. An example of the clustered result is demonstrated in Figure 3.7. We then select service instances using a round-robin like heuristic and allocate the instances to the power nodes from the top level of the power infrastructure

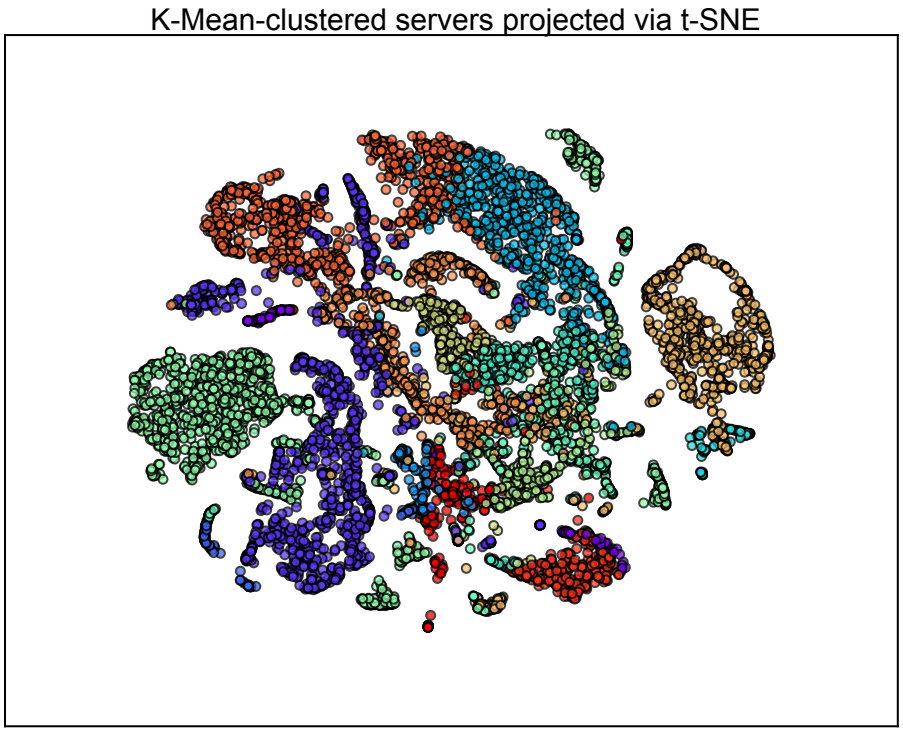


Figure 3.7: The production service instances in one of the suites of DC1 are embedded into the $|B|$-dimensional asynchrony-score space. k-means clustering is applied to classify asynchronous servers (shown in different colors). This figure shows the projected result onto a 2-dimensional space via t-SNE [93].

to the bottom.

For example, assume that we want to place a set of service instances I into the datacenter. We start from the top level l_0 and want to allocate service instances to q second-level nodes. The first step is to extract $|B|$ S-traces out of these servers. For each server, we calculate one asynchrony score against each S-trace, and have $|B|$ asynchrony scores for each service instance in the end. Each server is then considered as a data point in the $|B|$-dimensional space spanned by the asynchrony scores. We then apply k-means clustering to these data points and obtain a set of h clusters, denoted as $C = \{c_1, c_2, c_3, ..., c_h\}$. We configure h to be a multiple of q. Each of these clusters have the same number of instances. For each second-level power node, we iterate through all the clusters, and assign $\frac{|c_j|}{q}$ service instances from cluster j to that power node, and so on, until all the service instances are assigned to the second level power nodes. The process repeats and terminates when all the service instances are assigned to the last-level power nodes.

3.2.6 Adapting to workload changes

In datacenters, traffic and workload can change over time. Short-term workload uncertainties such as power spikes caused by traffic bursts are handled by commonly deployed emergency measures such as power capping solutions [150], and is out of the scope of this work. On the other hand, mid-term to long-term workload changes are what we care the most about. These changes are usually caused by the change of accessing patterns, which might gradually make the power efficiency of the current deployment suboptimal. Although, according to the past data in Facebook datacenters, significant changes rarely occur within months, we want to be able to identify when the current placement becomes suboptimal and apply incremental adjustment to it. To address this issue, our framework continuously records the I-traces and the S-traces, and dynamically re-evaluate the severity of the fragmentation problem by

monitoring the sum of peaks of power traces at each level of power infrastructure. When the placement becomes suboptimal with respect to the changing workload, we identify the power nodes with the most severe fragmentation problem (i.e., the node with the lowest asynchrony score or largest sum of peaks), and calculate a *differential asynchrony score* for every server. A differential asynchrony score is calculated between the I-trace of a service instance and the *averaged aggregate power trace* of a large group of server. We define

$$\bar{\boldsymbol{PA}}_{i,N} = \frac{\sum_{(j \in S_N \wedge j \neq i)} \bar{\boldsymbol{PI}}_j}{|S_N - 1|}$$

to be the averaged aggregate power trace of service instance i against power node N, where i is the service instance we are evaluating, and S_N represents the set of servers supplied by power node N; we define the differential asynchrony score of instance i against power node N to be

$$AD_{i,N} = \frac{peak(\bar{\boldsymbol{PI}}_i) + peak(\bar{\boldsymbol{PA}}_{i,N})}{peak(\bar{\boldsymbol{PI}}_i + \bar{\boldsymbol{PA}}_{i,N})},$$

we can then choose the service instance having the worst (lowest) differential asynchrony score, and swap it with some other service instance from another power node, if and only if that swap make the differential asynchrony scores higher at both of the two power nodes involved.

3.3 Exploiting power budgets with dynamic power profile reshaping

In the previous section, we explore how workload-aware service instance placement based on temporal power behaviors would help alleviating power budget fragmentation. Such mitigation of fragmentation allows datacenters to host more servers and

improve throughput. In this section, we further explore the opportunities of achieving additional throughput increase by carefully utilizing the extra servers.

Based on our investigation of production power traces, we design a **proactive dynamic power profile reshaping** approach, which includes two steps: (1) *history-based server conversion* and (2) *history-based proactive throttling and boosting* to further utilize the power headroom. In the rest of the sections, we first discuss the challenges of fully utilizing the unleashed power budget achieved by service placement. We then show how we leverage server conversion with *storage disaggregated servers* [74, 75] to improve the throughput by keeping these extra servers well utilized at all times. Lastly, we show how proactive throttling and boosting intelligently manage power budget allocation, allowing us to deploy extra conversion servers inside datacenters.

In the rest of the section, we denote *latency-critical* workload using **LC**, and *non-latency critical, throughput-oriented* workload using **Batch**.

3.3.1 Challenges

To utilize the unleashed power headroom we can add extra service-specific servers. This approach, although can improve throughput for the specific service, leaves throughput opportunities on the table. For example, assuming that we add LC-specific servers to a datacenter to use the unleashed power headroom and accommodate extra traffic. During the off-peak hours, even with the increased traffic, however, the original set of LC servers are likely to be sufficient to handle the burden without hurting the QoS of LC servers. In other words, newly added LC-specific servers will be underutilized during those hours. To address the low utilization issue, we want the set of "extra servers" to be able to host batch services to further improve batch throughput during off-peak hours.

3.3.2 History-based server conversion

3.3.2.1 Conversion with storage-disaggregated servers

Our insight is that the recently proposed storage-disaggregated servers [74, 75] is an ideal platform for solving the above issue. Storage-disaggregated servers are recently widely deployed in Facebook datacenters. In storage-disaggregated servers, the main storage components (i.e., Flashes) are separated from the compute counterparts (i.e., CPUs and memory). The compute nodes access the storage nodes over a high-bandwidth in-datacenter network instead of local PCIe links. This disaggregate approach incurs minimum overhead because the network access latency (microsecond-level) is small compared to disk access latency (millisecond-level) [74].

3.3.2.2 Benefits of server conversion

Using storage-disaggregated servers, we can design server conversion to fully utilize the newly-added servers during both peak and off-peak hours. Server conversion switches the service a server hosts, between LC service and batch service, adaptively based on the load. Server conversion with storage-disaggregated servers offers several advantages. First, it allows us to accommodate the increased LC traffic at peak hours by hosting only LC service and improve the Batch throughput during off-peak hours. Second, these storage-disaggregated servers allow us to maintain a better overall resource utilization while maintaining the data availability. This is one key advantage of using storage-disaggregated servers. Because data reside in their dedicated storage nodes, and is intact and still accessible by other Batch servers even when the server is converted to LC servers during peak time. Third, the server conversion process is low overhead and does not require time-consuming data migration. Last but not least, because server conversion does not require any OS reboots, the OS is always running, meaning that even during conversion, the converted servers are still controllable by

other runtime monitors, which ensures that the power safety is maintained during the conversion process.

3.3.2.3 Design of conversion policy

Inspired by prior proposals [91, 97, 17] which leverage workload colocation on leaf machines to improve node-level resource utilization, we design a *server conversion policy* to allow LC and Batch workloads to safely share the available power budget throughout the hierarchy of the power infrastructure. We are also inspired by previous works [161] and design our conversion policy by taking advantage of the clear patterns found in historical LC load data. The server conversion policy is designed to be applied on the set of conversion servers e_{conv} as follows: First, we learn the *guarded per-LC-server load level* from the historical data (training data), namely the load level of each server when LC achieves satisfactory QoS and define this load level as the *conversion threshold* (L_{conv}). During runtime, we continuously monitor the LC server load over each original set of LC servers. Based on the average load level, we distinguish two phases: a *Batch-heavy Phase* and a *LC-heavy Phase*. When the average LC server load over the original LC server is smaller than L_{conv}, this datacenter is in *Batch-heavy Phase*. When this average LC load increases to a level close to L_{conv}, our server conversion demands the conversion servers to be converted to LC instances, and we enter *LC-heavy Phase*. The threshold L_{conv} is also used to manage the load on each LC server. If any of the LC servers experiences a load higher than L_{conv}, then our server conversion process will stop sending queries to this server, and, instead, send the next query to other LC servers or a conversion server.

We can further maximize the throughput improvement by throttling the Batch clusters' power consumption during peak hours. Such throttling proactively creates additional power headroom during the peak hours, allowing us to house an additional set of conversion servers e_{th} in the datacenter. To achieve further throughput improve-

ment, we augment the previous policy to leverage this set e_{th} of conversion servers. We monitor the load of the original set of LC servers and of the LC servers in e_{conv}, comparing the load with the same conversion threshold L_{conv}. In this augmented policy, the definitions of *LC-heavy Phase* and *Batch-heavy Phase* remain unchanged. One distinction between this augmented policy and the previous policy is that, when the average LC load over the original LC servers and the LC servers in e_{conv} approaches to L_{conv}, we now first throttle the Batch clusters, and then it starts to convert servers in e_{th} into LC servers. Another distinction is that, during *Batch-heavy Phase*, we *boost* the performance of Batch servers to compensate for the loss of throughput caused by the throttling.

3.4 Evaluation

3.4.1 Experimental setup

In this work, we conduct our experiments using the power traces measured in three of Facebook's largest datacenters. All of these three datacenters are power supplied by the multi-level power infrastructure described in Section 3.1. Each datacenter consists of four suites and tens of thousands of servers. For every server housed in these three datacenters, we measure and log three weeks of power trace. The averaged instance power traces constructed by taking the average of the first two weeks of instance power traces serve as our training data. The third week of power traces serve as our testing data. We derive SmoothOperator's power-efficient instance placement and power profile reshaping policies based on the training data, and evaluate the benefit of each these two components using the testing data. The time interval of a week serves well as the unit of evaluation because, in large-scale user-facing datacenters, user traffic has strong day-of-the-week activity patterns [15, 126].

3.4.2 Results

In this section, we present evaluation results that demonstrate the effectiveness of SmoothOperator on improving the efficiency of power usage and on achieving higher throughput in the three target datacenters. Specifically, we present the following two studies: 1) We show that SmoothOperator's workload-aware service instance placement framework reduces the peak power at different levels of power nodes, mitigating the power budget fragmentation problem and allowing datacenter operators to host more servers under the same power budget. 2) We show that, with SmoothOperator's dynamic power profile reshaping policy, we improve the power utilization during non-peak hour, which further improves service throughputs.

3.4.2.1 Peak power reduction by workload-aware service placement

We start with the study that shows how SmoothOperator's workload-aware service instance placement reduces the peak power and mitigates the power budget fragmentation problem. As we described in the previous sections, the distinctive diurnal patterns of services and the service-level and instance-level heterogeneity provide abundant opportunities for improving the efficiency of power utilization in these datacenters.

Figure 3.8 presents how our framework reduces fragmentation using production power traces. In this figure, we apply SmoothOperator's workload-aware instance placement to the sub-tree rooted at a middle-level power node N, including node N and all the descendent power nodes and the service instances supplied by N. We demonstrate in the top graph in Figure 3.8 the power trace N. In the middle graph, we show the power traces of the children power nodes of N, one for each child, *before* we apply the workload-aware service instance placement on N. In the bottom graph, we show the power traces of N's children nodes *after* we apply our workload-aware service instance placement to N. Note that the power trace at N is

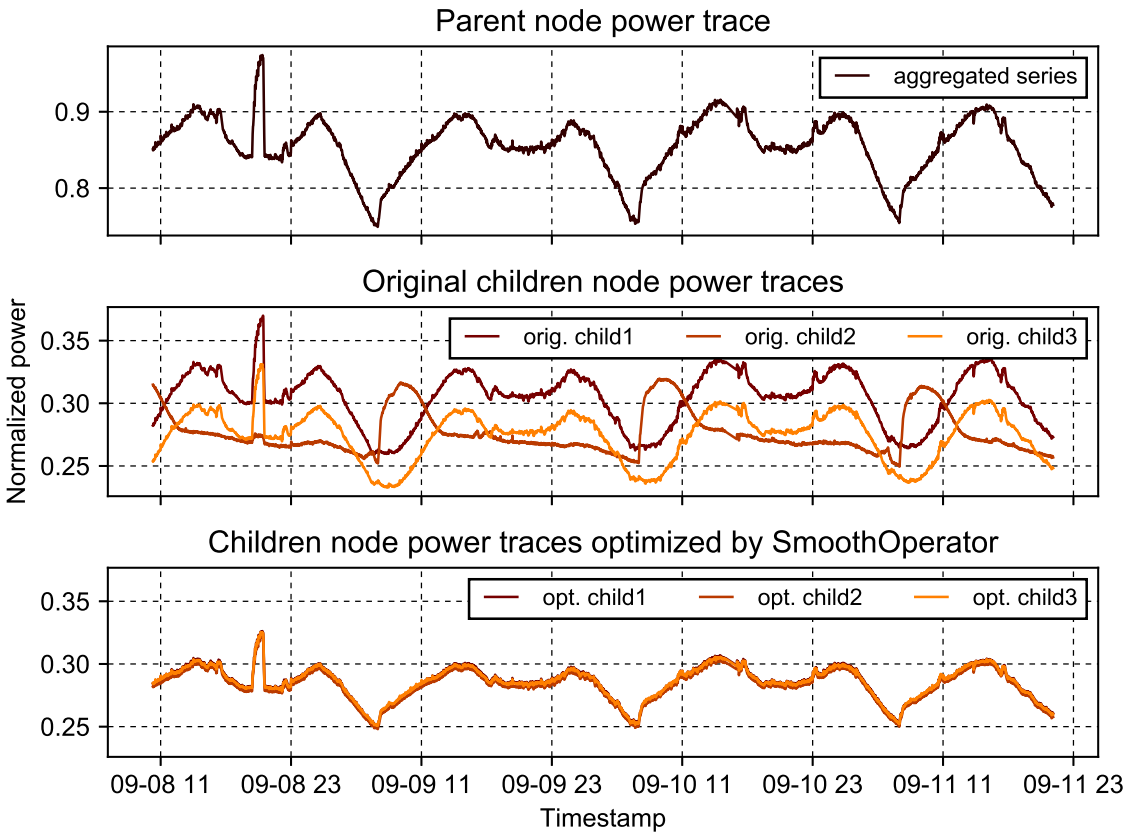


Figure 3.8: The comparison between the children power trace generated by the oblivious and workload-aware placement in a production suite of DC1. The workload-aware placement generates smoother power traces, greatly alleviating fragmentation. Power peaks are reduced at the child node thus more servers can be supported at each node.

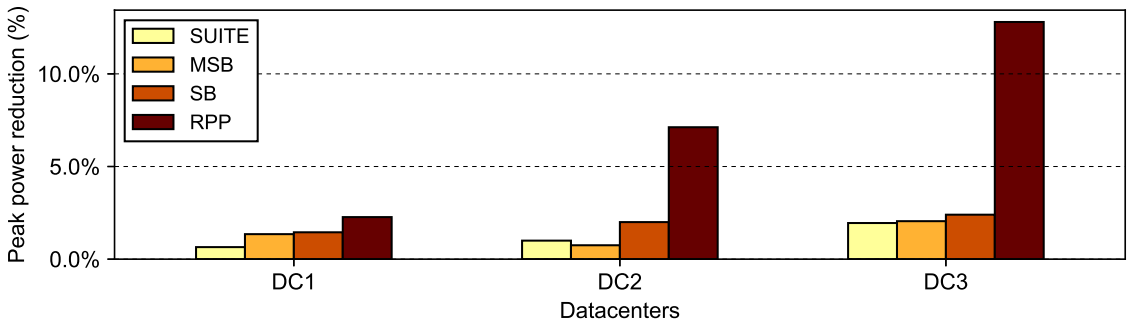


Figure 3.9: The peak-power reduction achieved at each level of the power infrastructure in the three datacenters under study. There is significant peak reduction at RPP level, which directly translates to the percentage of extra servers that can be hosted.

not changed by SmoothOperator because, in the example, our placement policy does not move service instances into or out of the subtree rooted at N. This figure shows that SmoothOperator, with the placement step alone, makes the power traces of the children power nodes less varying and more balanced, and reduces the peak power of the children power nodes.

Peak power reductions achieved at different levels of power infrastructure

We present the peak power reduction at all levels in all three datacenters in Figure 3.9. Recall from Section 3.1 that, for a same power delivery tree, the sum of peak powers of power nodes at a certain level is an important indicator of the severity of power budget fragmentation at that level. As shown in Figure 3.9, we find our workload-aware service instance placement can reduce the RPP-level peak power by 2.3%, 7.1% and 13.1% for the three datacenters, respectively. At higher levels, SmoothOperator achieves less significant reduction. The reason is that each of these higher level power nodes *indirectly supply* a group of up to thousands of service instances with higher degree of heterogeneity within them than the leaf nodes. Note that, however, the leaf power nodes suffer from fragmentation significantly. In other words, peak power reductions at the lowest level are of the utmost importance, as servers can only be directly power supplied by the low-level power nodes. These reductions translate to the proportion of extra servers allowed to be housed under

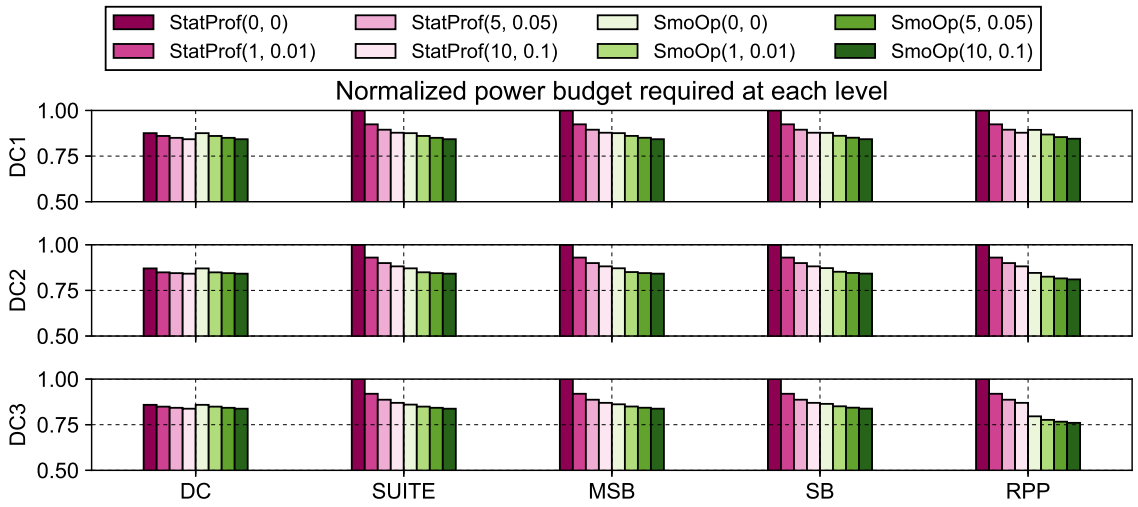


Figure 3.10: Required power budget achieved by previous work and SmoothOperator. StatProf(u, δ) refers to the result of previous work with a degree of under-provisioning u and a degree of overbooking δ. SmoOp(u, δ) refers to the SmoothOperator counterpart.

the same power infrastructure.

While we can extract power headroom in DC3, the benefit we get in DC1 is smaller. The reasons are two-fold: First, the degree of heterogeneity among instances power traces found is DC1 is much smaller than that in DC3. Second, due to the lower degree of heterogeneity, the baseline (original) placements in DC1 suites are more balanced compared to DC3. For DC3, synchronous service instances are largely placed under the same sub-trees of the power infrastructure in the original placement, allowing us to achieve improvement.

We compare our approach to a previous work [54]. This work aims to optimize the provisioning of the capacity of power nodes in datacenters. It models power pattern of instances and power nodes as cumulative distribution functions (CDFs), and relies on leveraging these probability distributions to aggressively under-provision and overbook power nodes.

For example, to power supply a set of service instances M, this previous work models the power profile of each instance i in M as a c_i and defines a *degree of*

under-provisioning u. The budget of the power node that supplies M will be set to $\sum_{i \in M} c_{i,u}$, where $c_{i,u}$ denotes the $(100-u)$-th percentile power of instance i's power profile c_i. This work also recognizes that, at datacenter-level, they can take advantage of the heterogeneity among the instances with overbooking, and defines a *degree of overbooking* δ, which further reduces the datacenter-level provisioning requirement. Suppose the datacenter-level power capacity was $\sum_{i \in dc} c_{i,u}$; with δ, the capacity can be further reduced to $\sum_{i \in dc} c_{i,u}/(1+\delta)$.

The comparison can be found in Figure 3.10, which shows the power budget provisioning required by the three datacenters after applying the two approaches. Since the under-provisioning and overbooking techniques used in the prior work is independent with our techniques, in our experiment, we also add several configurations of SmoothOperator (`SmoOp`) in which under-provisioning and overbooking are allowed. We denote `StatProf(`u`, `δ`)` as the configuration of previous work with a degree of under-provisioning u and a degree of overbooking δ, and `SmoOp(`u`, `δ`)` as the SmoothOperator counterpart. Note that, with such notation, SmoOp(0, 0) represents using SmoothOperator alone without any under-provisioning nor overbooking.

We can see that SmoOp(0, 0) achieves ¿12% of reduction in the required power budget provisioning in all cases, and it achieves higher level of improvement over the prior work as we go down the hierarchy of power infrastructure. When compared to StatProf, across all the levels, SmoOp(0,0) almost always outperforms or is on par with the most ambitious StatProf (i.e., StatProf(10, 0.1)). If we allow SmoothOperator to under-provision and overbook, across all levels and datacenters, it always requires less power budget provisioned than the StatProf counterpart, by up to ¿10%. For instance, in DC3, while StatProf(10, 0.1) achieves only 13% of reduction, SmoOp(0, 0) achieves 20% and SmoOp(10, 0.1) achieves 24%.

These results show that, by intelligently leveraging temporal heterogeneity among instance power traces, SmoothOpeartor lowers the peak of the power profiles at power

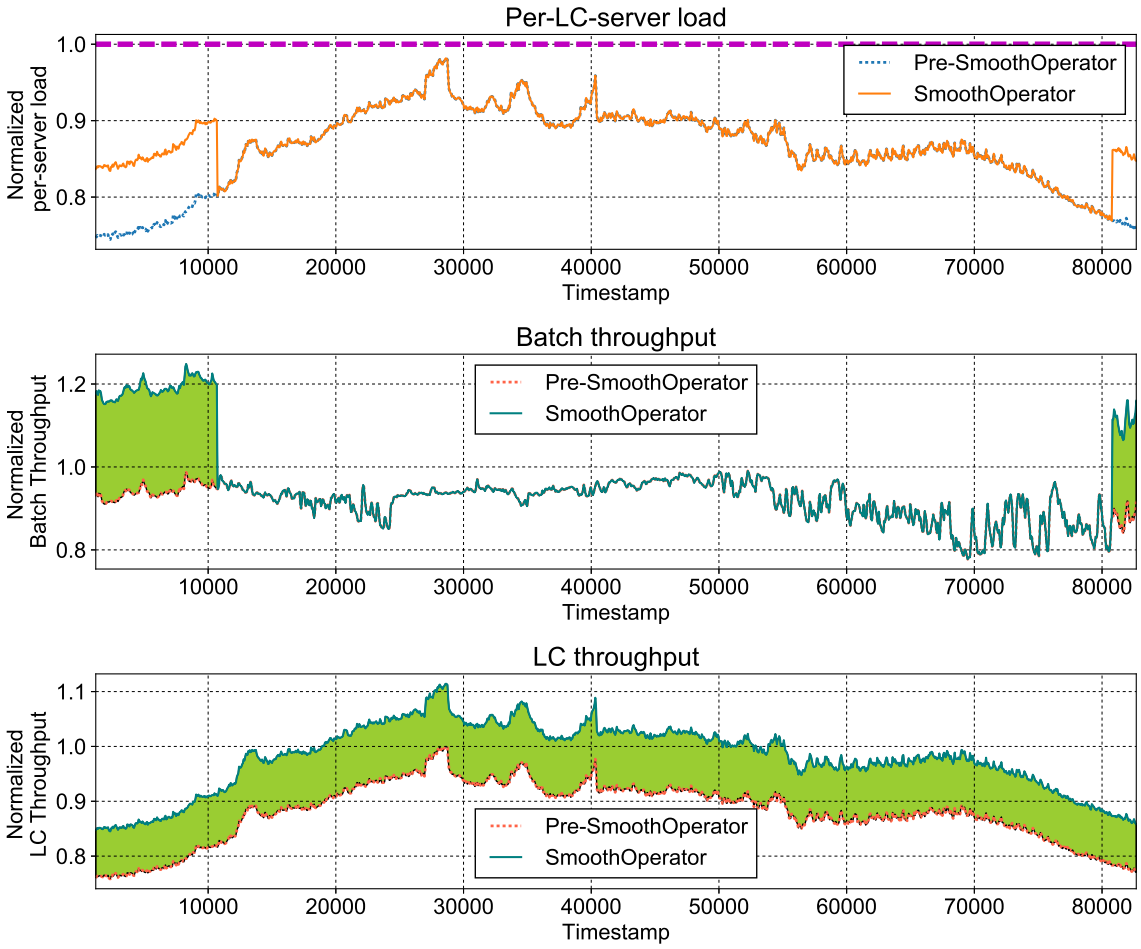


Figure 3.11: Server conversion's impact on per-LC-server load, LC and Batch throughput.

nodes, *creating* the extra headroom that was not available to the prior work. Moreover, since SmoOp(0, 0) always outperforms StatProf(10, 0.1), we conclude that the benefit we get from SmoothOperator does not need to rely on probabilities, implying higher level of safety guarantee at power nodes.

3.4.2.2 Benefit of dynamic power profile reshaping

In this subsection, we present evaluation results demonstrating the benefit of using dynamic power profile reshaping to utilize the power budget unleashed by SmoothOperator's workload-aware service instance placement.

Impact of server conversion We present a segment of our experiment (Figure 3.11)

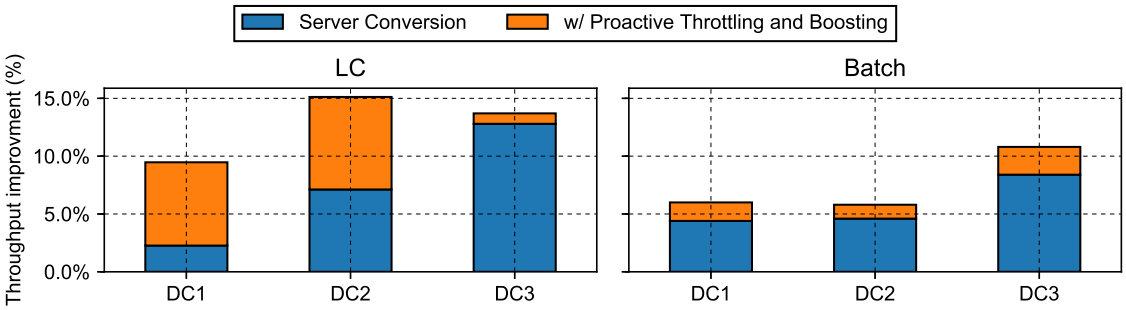


Figure 3.12: The breakdown of throughput improvement of LC and Batch services.

to highlight the server conversion's impact on Batch throughput, LC throughput, and the load per LC server. In this example, the DC optimized by the workload-aware service instance placement has 11% of extra power headroom to accommodate extra traffic. If we add only LC-specific servers, this datacenter can achieve 11% extra LC throughput. If we use conversion servers to fill in the power gap, we gain extra benefit for Batch services. As shown in the top subgraph in Figure 3.11, during Batch-heavy Phase, the per-server load for LC-servers is low and the original set of LC servers are underutilized. During this time, we do not need extra computing power to handle the incoming LC queries. As long as the LC servers are under a specified guarded load level, the added conversion servers can be converted to Batch service instances to perform Batch workload, as shown in the middle subgraph of Figure 3.11. On the other hand, during LC-heavy phase, the original set of LC servers do not have the capacity to handle the increased LC traffic. This is when conversion servers need to kick in to help reduce the per-server load for LC servers. In the top subgraph of Figure 3.11 we see such an impact.

Figure 3.12 presents the throughput improvement achieved by both server conversion and proactive throttling and boosting policies throughout the entire experiments. It shows that, with server conversion alone, we are able to utilize the 13% unlocked power budget to trade for up to 13% LC throughput plus 8% Batch throughput at the same time.

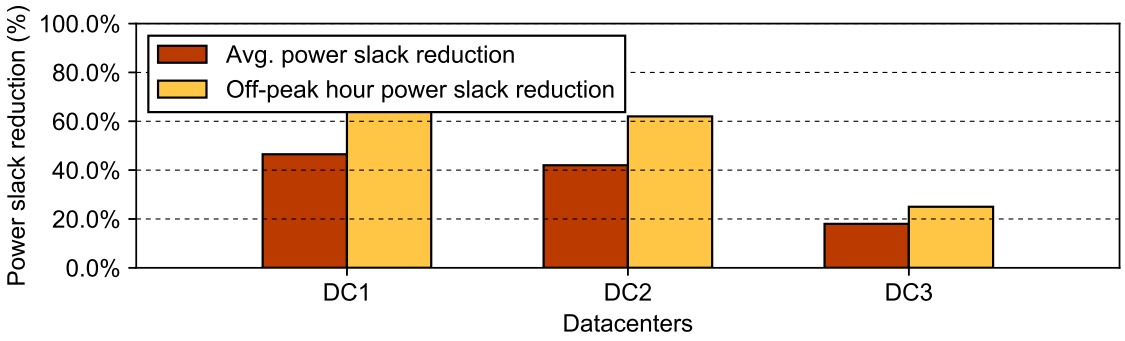


Figure 3.13: Average power slack reduction and off-peak phase power slack reduction achieved at three of Facebook's production datacenters.

Impact of proactive throttling and boosting In this section, we evaluate the benefit we obtain from applying proactive throttling to the Batch instances. The throttling and boosting policy proactively throttles the Batch services to increase the power budget allocation for LC services during LC-heavy Phase; during the Batch-heavy Phase, it only increases the allocation and boosts the performance of Batch instances, but does not throttle the LC services.

Figure 3.12 demonstrates the extra throughput improvement achieved by the proactive throttling and boosting policy on top of server conversion. The improvement of Batch throughput here is small, 1.6%, 1.2%, and 2.4%, for the three datacenters, respectively. This is as expected because the gain brought by the extra conversion server barely compensates the loss caused by aggressive throttling during LC-heavy Phase. On the other hand, we largely improve the capacity of the LC service in 2 of the 3 datacenters during the peak hours. The extra improvement of LC throughput is 7.2%, 8%, and 1.8% for DC1, DC2, and DC3, respectively, which translates to a capacity gain that can accommodate multi-millions extra queries per second.

Power slack reduction In Figure 3.13, we present the result of average power slack reduction and off-peak hour power slack reduction of the three datacenters. The reduction of power slack means that the power budget available during off-peak hours is used to do more work. From Figure 3.13 and Figure 3.12 we find that the benefit

gained in DC3 is smaller compared to the other two datacenters. The reason of this is that DC3 has a large proportion of LC service instances among the top power consumers compared to the Batch type counterpart. Therefore fewer Batch service instances can be throttled for LC to borrow power budget from, limiting the improvement of Batch throughput. In other two datacenters, power slack reduction are achieved. As shown in Figure 3.13, the dynamic power profile reshaping achieves 44%, 41%, and 18% average power slack reduction, respectively in the three datacenters.

3.5 Summary

In this work, we investigate three of Facebook's datacenters and aim to solve the power budget fragmentation problem found in them. We leverage the knowledge of the temporal heterogeneity of power consumptions of different workloads, and apply our workload-aware service instance placement technique to unlock the wasted power budget, and evaluate the effect of server-conversion-based dynamic power profile reshaping runtime in production environment. Our result shows that we are able to host up to 13% more servers in production environment by applying our placement technique, without making modification to the underlying power infrastructure. To this end, we reduce up to 44% of average power slack inside datacenters. Without incurring significant burden on latency-critical servers, we achieve up to 13% plus 8% throughput improvement for latency-critical service and batch service, respectively, by using server conversion alone; or 15% and 11% throughput improvement with the help of DVFS.

CHAPTER IV

Pinpointing and Reining in Long Tails in Warehouse-Scale Computers with Adrenaline

In this chapter, we present Adrenaline – our effort for solving power and energy inefficiency at the server level by using quick voltage/frequency boosting. First, the high-level design principle of Adrenaline is presented, then we present the energy-saving opportunity we identify in datacenter workloads. We then introduce the details of the design of Adrenaline and provide an empirical evaluation of Adrenaline's impact on improving datacenter performance with low power and energy usage.

4.1 The Design Principle of Adrenaline

Adrenaline aims at leveraging knowledge about query-level characteristics and the emerging class of fine-grain (10's of nanoseconds) voltage boosting (i.e., *quick boosting*) techniques to achieve significant tail latency improvement while still being highly energy efficient. A key insight underlying Adrenaline is that only a subset of queries need boosting to pull in the tail latency. Two key factors determine the subset that should be prioritized: 1) queries that fall in the tail distribution; 2) queries that can benefit most from the boosting. Our design of Adrenaline tries to address these two factors. However, several open questions remain to realize this approach,

including:

- *Investigating whether tail queries are amenable to frequency/voltage boosting.* If tail queries in representative OLDI workloads are not bottlenecked on computation, V/f boosting will not be effective in reducing their completion time and would not be able to pull in the tail.
- *Determining whether tail queries are predictable.* If the queries that push out the long tail share common characteristics, we can use these characteristics to develop per-query indicators to pinpoint queries for boosting.
- *Identifying an effective system design to pinpoint tail queries and precisely boost them.* Considering the query latency for many OLDI services is in the range of milliseconds and microseconds [88, 12], to realize quick, pinpointed boosting of tail queries, any purely reactive approach might spend most of the time waiting and monitoring, rendering the advantage of quick boosting useless. Therefore, an intelligent, proactive mechanism must be in place to enable the identification and boosting of likely tail queries.

4.2 Motivation and Opportunities

We begin by examining the query latency distributions of two representative web services and exploring how frequency impacts query latency. A key insight underlying Adrenaline is that only a subset of queries need boosting to pull in the tail latency. Two key factors determine the subset that should be prioritized: 1) queries that fall in the tail distribution; 2) queries that can benefit more from the boosting. To understand how we might identify queries that exhibit these factors, we characterize the query distributions of Web Search and Memcached workloads on a state-of-the-art Intel Xeon server (described in Section 5.5).

Figure 4.1 presents the cumulative distributions of request latency for the three most common request types for Memcached (`SET`, `GET` and `DEL`), collected at seven

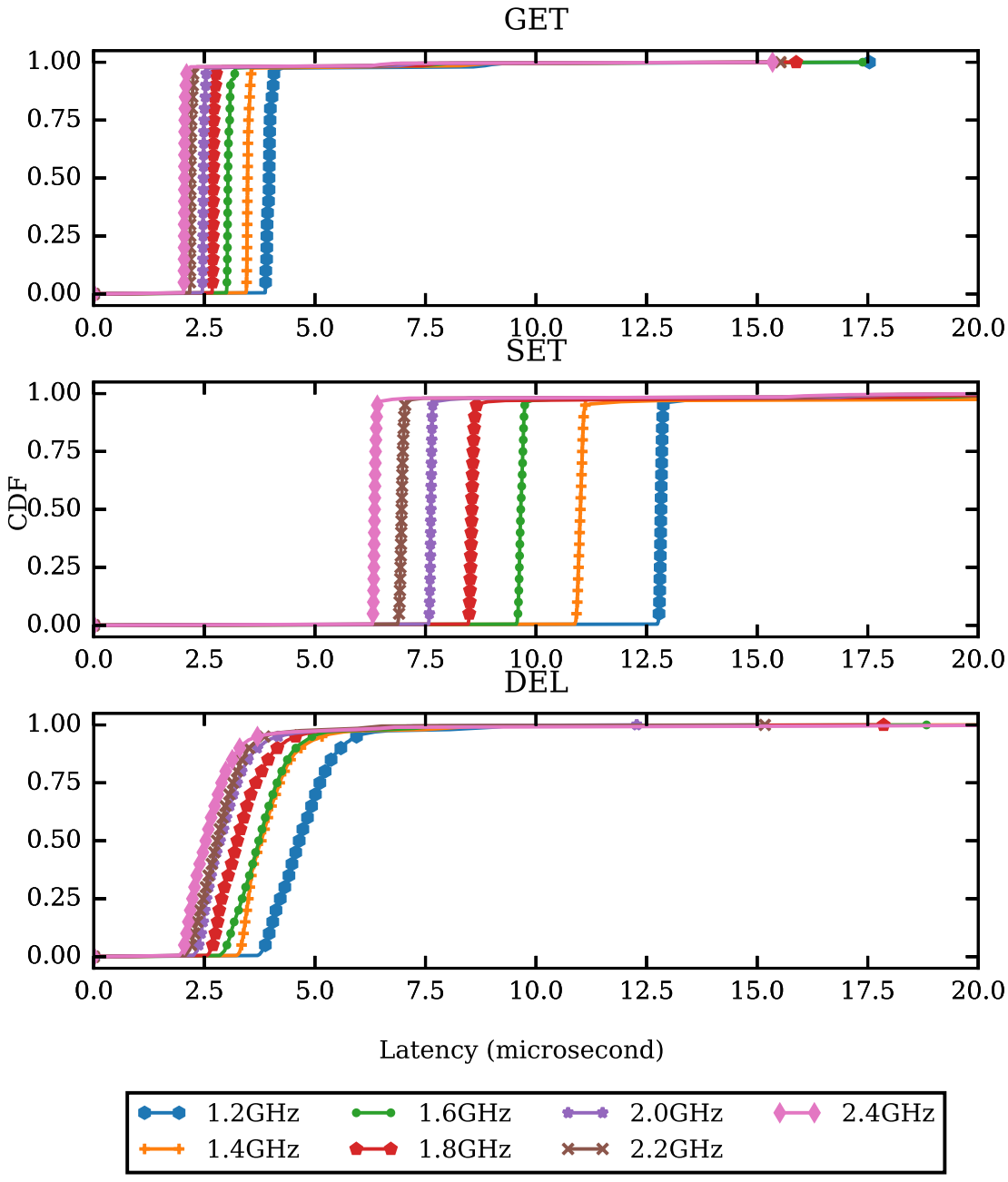


Figure 4.1: The cumulative distributions of query latency in Memcached for `GET`, `SET`, and `DEL` requests.

CPU frequency steps ranging from 1.2 GHz to 2.4 GHz on the Intel Xeon server. As shown in the figure, although all three query types have long tails, the latency distributions of these three types and how they are affected by frequency scaling vary. `SET`s' request latency is in general around 2x greater than `GET`s or `DEL`s, indicating that the `SET` requests may contribute more to the tail latency, especially at low to medium load levels. In addition, higher frequency can significantly improve `SET`s' latency. Increasing the frequency from 1.2 GHz to 2.4 GHz improves `SET`s' 90-th percentile latency from 13μs to 7μs. This indicates that to maximize the benefit of tail reduction using voltage/frequency boosting under a power budget, boosting `SET` requests should be prioritized.

In contrast to Memcached, Web Search (Nutch [46]) does not have multiple query types that can be directly used to classify queries. After investigating multiple query characteristics and the effectiveness of using those characteristics to predict the query latency distribution and the impact of frequency of scaling, we have identified that the query length (the number of search key words in a Web Search query) is a fairly effective indicator. Figure 4.2 presents the cumulative distributions of query latency for three different query lengths at seven frequencies, focusing on the tail part of the latency (beyond 85-th percentile). As shown in the figure, short queries (queries with fewer, e.g., 1-5, search key words) in general experience longer latency than long queries (queries with more search terms). Whereas it may seem counter-intuitive that adding terms reduces query latency, Nutch returns only documents that contain all search terms. Hence, additional terms reduce the number of documents that must be considered in scoring. In addition, the latency of short queries is much more improved when we increase the frequency, compared to the medium and long queries. For example, the 95th percentile latency of queries with 1-5 keywords (short) is around 1100ms at 1.2GHz and lower than 700ms at 2.2GHz. On the other hand, the queries with 11-18 search keywords (long) are not affected by the frequency as much. This

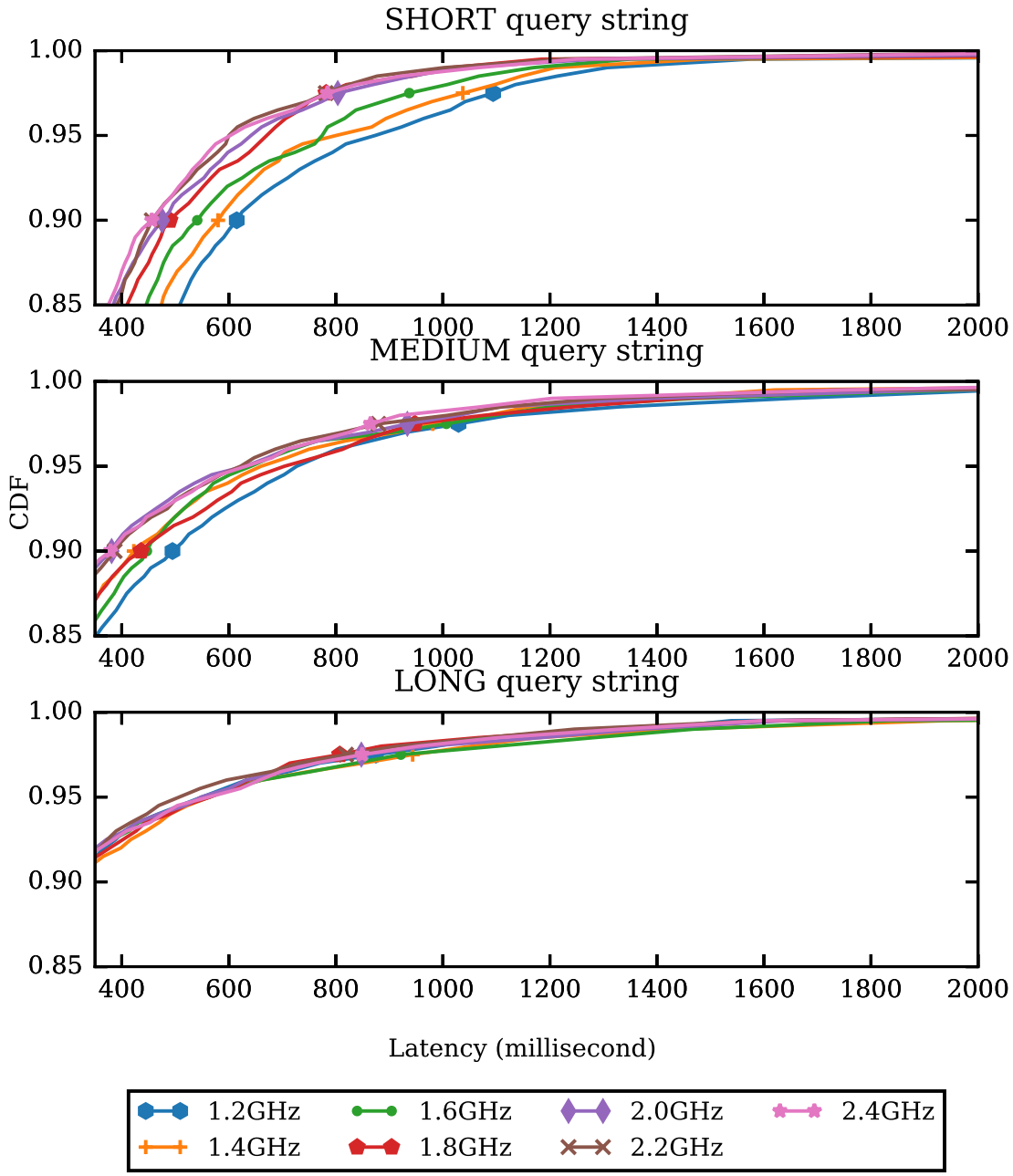


Figure 4.2: The cumulative distributions of query latency in Web Search for queries with different numbers of search keywords.

indicates that queries with fewer terms (short) should be prioritized for boosting to pull in the tail.

In summary, Figures 4.1 and 4.2 demonstrate that per-query indicators (e.g., query types and query properties) can help pinpoint a subset of queries that contribute more to the tail and/or more impacted by the frequency and thus need to be prioritized to effectively pull in the tail. Based on this, we propose query-level boosting. Figure 4.3 illustrates the difference of our system, Adrenaline, compared with no boosting and traditional coarse-grain DVFS boosting. As illustrated in the figure, coarse-gain approaches apply V/f boosting to a sliding window, boosting all queries within a window. Adrenaline takes advantage of fast boosting (sub 20ns) and uses per-query indicators to pinpoint individual queries that should be boosted (for example, `SET`s as illustrated) to conduct query-level boosting only for these queries to achieve effective tail reduction with high energy efficiency.

4.3 Quick V/f Boosting: An Enabling Technology

There are existing technologies that enable fast voltage transitions. Per-Core DVFS was a technique explored by Kim et al. [72, 71]. Per-Core DVFS integrates a voltage regulator on-die for each core in the system, allowing individual control and nanosecond scale voltage transition times. The Per-Core DVFS technique comes at the expense of on-chip inductors and reduced regulator efficiency. Intel's Turbo-Boost [28] enables microsecond scale voltage transitions to allow, for example, the system to exceed thermal budgets for short periods of time. TurboBoost also includes an integrated Power Control Unit (PCU) which makes boosting decisions in hardware based on sensors and other hardware performance counters, eliminating the long latencies associated with OS management. Recently, Godycki et al. have introduced Reconfigurable Power Distribution Networks (RPDN) [53] as a method to improve voltage transition times with the use of a configurable on-chip switch-cap

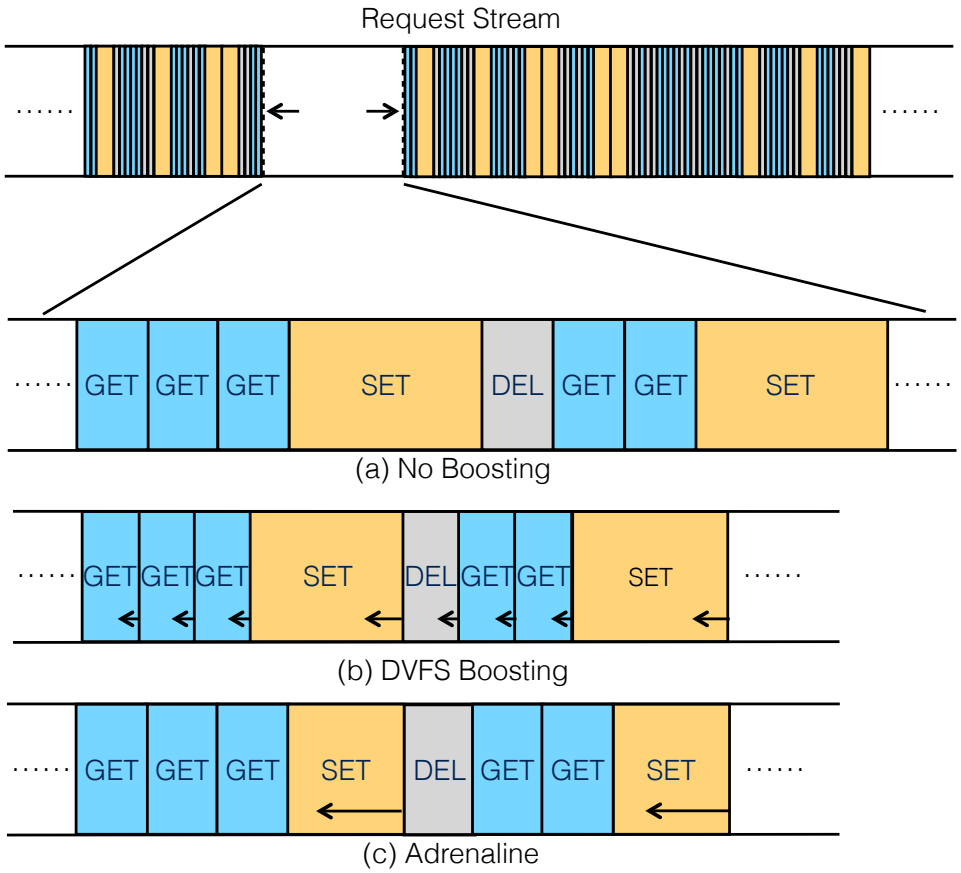


Figure 4.3: Adrenaline's Query Level V/f scaling vs. Coarse-grain Sliding Window based V/f scaling.

based voltage regulator. Finally, Shortstop [119] and Booster [106] use dual-rail voltage systems to enable fine-grain boosting. They use off-chip voltage regulation for better efficiency and to remove the need for on-die inductors.

For this work we evaluate Shortstop as the voltage regulation methodology due to its short transition latency, lack of on-die inductive elements, and better regulator efficiency. However, the other approaches could be adapted as well to support Adrenaline. For example, the Power Control Unit (PCU) of Turbo Boost could be augmented to accept query indicators to support the Adrenaline decision engine at the expense of a longer voltage transition latency.

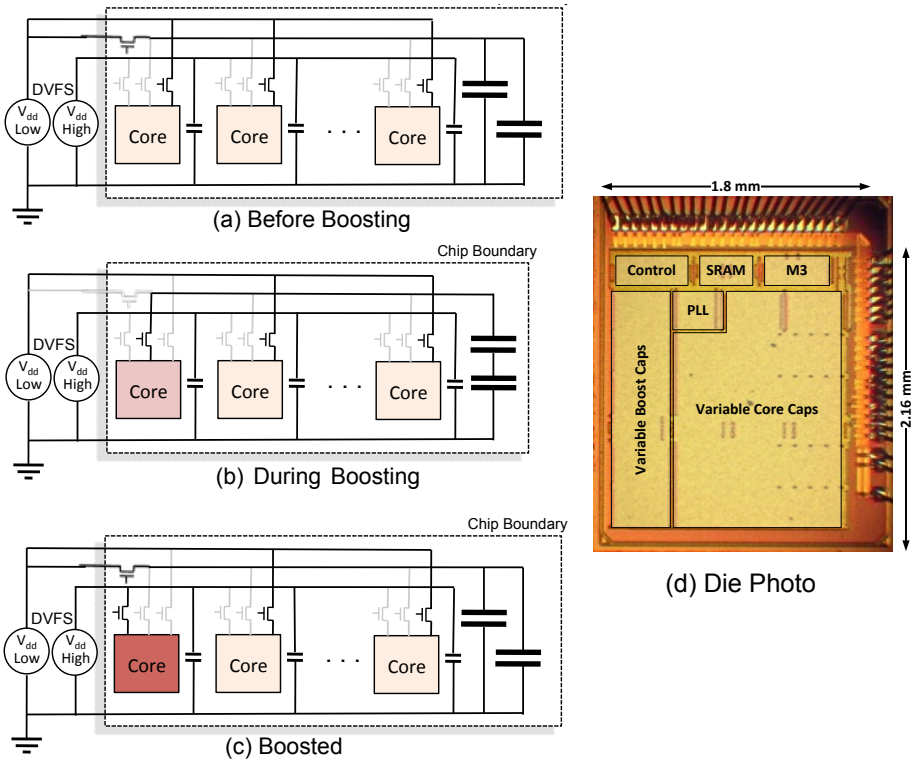


Figure 4.4: Shortstop's Dual-V_{dd} chip configurations. (a) shows normal operation, where all cores are connected to the low voltage network and the cap networks are in parallel. (b) shows the cores in boost transition where the core boosting is connected to the output of the serially connected cap network. (c) shows the system once the transition stabilizes where the cap network returns to parallel, and the boosted core runs off the external high voltage. (d) shows the die photo of the 28nm Shortstop test chip [119].

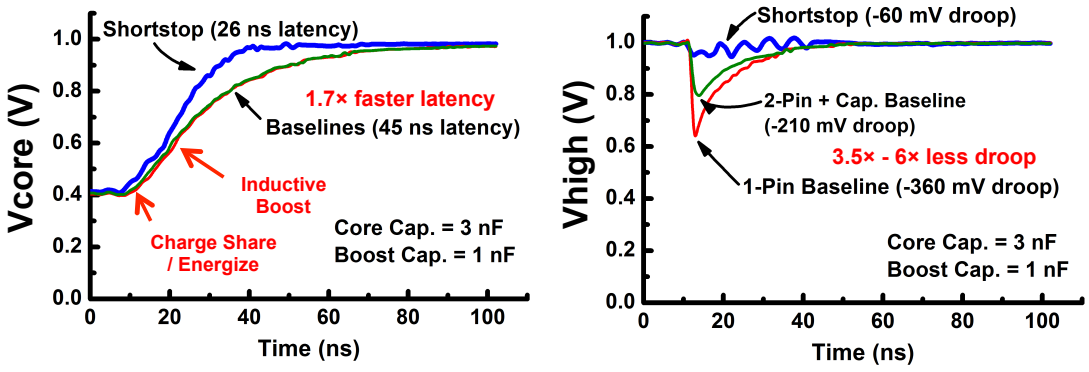


Figure 4.5: Measured chip results [119] of Shortstop compared to a standard dual-rail baseline.

4.3.1 Shortstop

Shortstop [119] is a dual-V_{dd} system with two distribution networks for V_{dd} supplied by two independent external voltage sources, V_{ddHigh} and V_{ddLow}. In addition, there is an internal V_{boost} supply to aid in the transition of a core from the low to the high supply. Each core in the system is connected via multiple power gating transistors (shown as a single transistor in the diagram) to either the V_{ddHigh}, V_{ddLow}, or V_{boost} voltage rails. Decoupling capacitors (decaps) are placed between the high supply network and the ground node to reduce ripples on the node during transitions. In addition the V_{boost} supply has a set of reconfigurable decoupling capacitors to aid in transitioning the core quickly. This system keeps the voltage regulators off-chip removing the TDP overheads required for on-chip conversion.

Using Shortstop has several advantages. First, since the boosting decaps are on chip, they can act quickly and, through charge re-distribution, provide for a rapid transition. Second, since the boosting decaps are shared by all the processors, their area overhead is amortized over all the cores. In addition, while the boosting network does require the distribution of a third supply rail (V_{boost}), this rail does not need to have a high level of signal integrity, meaning it can be more sparse. The overall overhead of adding a boosting rail with reconfigurable decaps is 11%. When considering advanced technologies, such as deep trench capacitors [146], this overhead can be reduced to less than 5%. Third, since the boosting network brings the voltage of the transitioning core to nearly V_{ddHigh}, the voltage droop on V_{ddHigh} is not nearly as large as when no boosting network is used. Extra decaps on the high supply further suppress the droop to a level that is acceptable. The area overhead of the V_{ddHigh} rail is just 5%. So the overall area overhead of Shortstop is ∼10-16%. Finally, Shortstop can be used to perform both fine- and coarse-grain voltage control. Through boosting Shortstop provides nanosecond scale voltage transitions to adjust to fine-grain changes in behavior (query level) and can adapt the off-chip voltage regulators over

longer periods of time to adjust to coarse-grain changes in workload behavior (request rate level).

Shortstop was fabricated and tested in a 28nm technology by Pinckney et al. [119], proving the feasibility of the approach. Figure 4.5 provides a summary of the measured data on transition times of that chip. The baseline for comparison was a standard dual-rail approach. Shortstop provides nanosecond scale transitions that are 1.7x faster with a 3.5-6x smaller voltage droop. The original paper also provides results showing the scalability of the approach across different core sizes/capacitances.

4.4 Adrenaline Framework

This section describes the design of the Adrenaline runtime system, which takes advantage of the fast V/f switching capability of the Shortstop circuit to rein in tail latency by boosting precisely those queries that contribute to the tail.

Adrenaline Overview

An overview of the Adrenaline runtime system is presented in Figure 4.6. Adrenaline is composed of two components, a decision engine and a load monitor. First, the decision engine chooses from between the low or high V/f settings at the individual query level to boost the speed of precisely those queries that contribute to tail latency, based on the query characteristics analyzed by the equipped query identifier, reducing the latency of those queries to shorten the tail of the query latency distribution. This is in contrast to conventional V/f scaling techniques that monitor the query load and distribution over long timescales (usually several seconds) and adjust the V/f settings for large clusters of queries.

Second, Adrenaline's load monitor performs lightweight accounting on incoming queries to measure the load to the system (e.g., to measure queries per second), which in turn is used to drive a coarse-grain tuning policy that changes the supply voltage parameters to the off-chip voltage regulators. This gives Adrenaline the capacity to

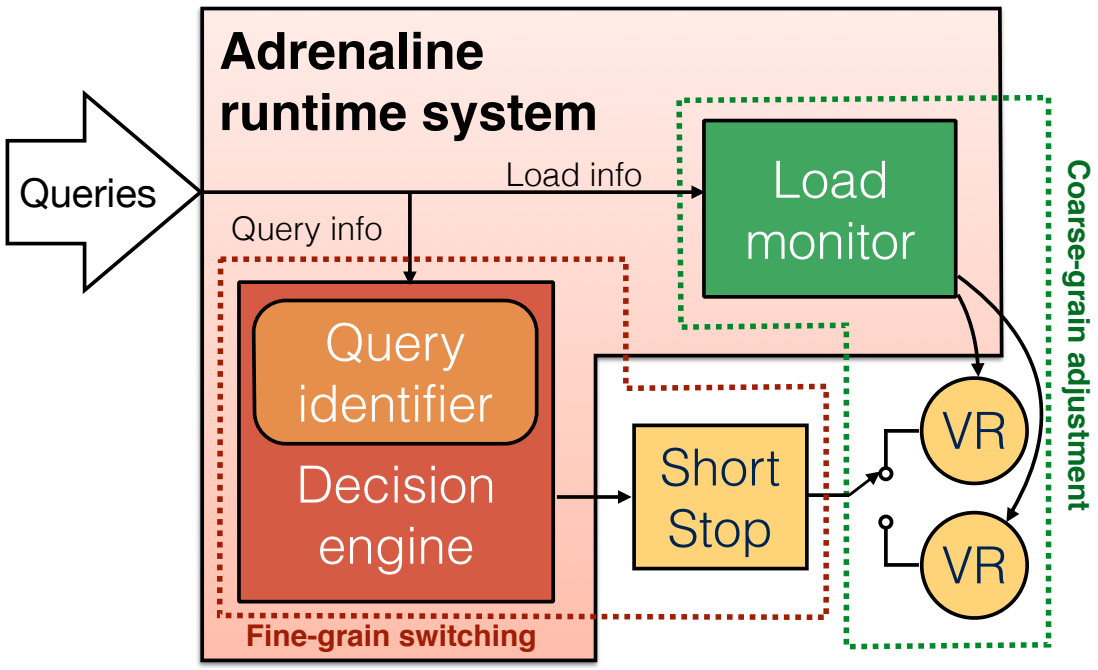


Figure 4.6: The Adrenaline framework. Adrenaline makes fine-grain boost decisions based on the characteristics of each incoming query, and controls the Shortstop circuit to switch between the high/low voltage rails (VRs) quickly. Meanwhile, Adrenaline also monitors long-term loading information, and makes proper adjustments to the voltage on the high/low VRs periodically.

adapt the Shortstop circuit to address longer-term (milliseconds or longer) changes in the characteristics of the query profile, such as fluctuations in user demand (queries per second) or the type and composition of queries.

4.4.1 Decision Engine

The purpose of the decision engine is to rapidly identify the queries that should be boosted to have the largest impact on reducing tail latency. First, queries that are amenable to boosting because they are compute-bound but are not in the tail can have a significant impact on tail latency via the indirect mechanism of reducing queuing delay for queries that are in the tail of the latency distribution. Second, queries that are themselves in the tail of the distribution have a direct impact on tail latency. We therefore design the decision engine to use the following two guiding principles to determine which queries to boost:

1. **Boostability** – only those queries that are amenable to boosting should be boosted. Power spent boosting queries that do not speed up is wasted. Furthermore, queries that are more amenable to boosting confer a larger benefit to subsequent queries that may be headed for the tail of the distribution.

2. **Long-running Queries** – as long as they are boostable, queries that are destined to be in the tail of the latency distribution have the most direct impact on tail latency and therefore should be boosted.

These characteristics may be difficult to directly reason about, particularly at the short timescale necessary to boost a query quickly after arriving at the host server. Fortunately, the presence of one or both of these characteristics are often indicated by other easily identifiable proximate characteristics. Such characteristics include:

- **Query Type** – different query types, which are easily identifiable as part of the query metadata, often have very different latency profiles. For example,

Figure 4.1 shows that Memcached `GET` queries have around a third of the latency of `SET` queries.

- **Query Composition** – the same type of query may have other easily identifiable characteristics that correlate well with query latency. For example, Figure 4.2 shows that Web Search has very different latency profiles for queries with different number of words.

4.4.2 Defining the Boost Policy

Query type and query composition are used to develop policies that determine when the Shortstop circuit changes rails, lowering and raising the supply voltage at the query level to bring down the latency of critical queries. These policies are developed offline using a short, targeted profiling phase on the application to characterize the relationship between query latency and the high-level characteristics of the query.

Adrenaline's boost policies are guided by how different types of requests can affect the tail the most under different V/f settings. For each application we first analyze the behavior of requests across varying characteristics, load levels, and chip V/f settings. The characteristics chosen for the analysis are particular to the application under study, and are chosen such that they are characteristics that can be rapidly identified (e.g., by checking a few bits in the header, or the size of the application query packet). For Memcached, we characterize requests across the different query types: `GET`, `SET` and `DEL`. For Web Search we characterize across a series of lengths, which correlates well to the size of the query packet and the latency of the query. For Memcached, we find that `SET` queries are highly amenable to boosting the bulk of the high-latency queries. For Web Search, we find that queries with fewer than 6 keywords are similarly amenable to boosting and have high latency.

Adrenaline also watches the condition in which a query tend to fall in the tail of the latency distribution, even if that query does not belong to the types mentioned

above. Adrenaline's policy is to keep track of the time a query has spent in the system, and check if it exceeds a predefined **boost threshold**. When the threshold is reached, the query is considered *long-running*, and has a high probability to end up falling in the tail; hence, Adrenaline will make boost decisions for this query. We found by experiment that setting the threshold to 0.5x of the QoS target of the benchmark gives us good results.

4.4.3 Rapidly Identifying Query Characteristics

Adrenaline is implemented as a runtime engine in the first layer of software that processes incoming request packets to take advantage of features of advanced NICs and low-latency networking stacks, such as OS-bypass, zero-copy, and direct cache access. These features facilitate extremely low latency packet delivery to the Adrenaline runtime engine, which can then rapidly examine packet headers to make a per-query boost decision; prototype systems have demonstrated packet delivery latencies below 1.5us and commercial offerings from vendors like Mellanox have similar performance characteristics [127]. In addition, by fixing the header lengths of the link, network, and transport layers, even shorter times can be achieved. In the case of Memcached the request type field can be attained by inspecting the corresponding bits in the application layer of a binary encoded Memcached packet. For the Web Search workloads a table of sequence numbers and corresponding packet sizes can be used to determine the total request length, which is used as a proxy for the size of the search.

4.4.4 Boosting and Unboosting the Core

Once a query type is identified, the Adrenaline runtime decision engine makes a boosting decision based on the profile characteristics obtained from the analysis of the request behavior. If a decision to boost a core not already boosted is made, the runtime signals the Shortstop hardware to insert the core into a boosting queue.

Because Shortstop only allows one core to transition at a time, the queue is serviced in a FIFO fashion. Note the worst case time spent in the queue will be short, as the transition times for Shortstop are on the order of 10's of nanoseconds and the number of cores on a chip is relatively small. If a decision to boost a core is made while the core is already boosted, the Adrenaline runtime tracks the additional request. Once all the requests finish, the Adrenaline runtime signals Shortstop to unboost the core.

4.4.5 Clock Distribution

Clock distribution is another key consideration in this design. Our proposed solution is to distribute a chip-wide clock at the high frequency, but at low voltage. At each node the clock is divided down to the required frequency before entering each core. In a typical system the majority of the clock power is consumed at the bottom of the tree (i.e., flip flops) meaning that running the clock globally at high frequency has minimal impact on the total power. In boost mode the local clock tree remains at low voltage and is level converted to V_{ddHigh} only in the last driving stage. With this approach, clock tree synchronizing mechanisms such as delay-locked loops (DLLs) would not need to re-lock during boosting transitions since the voltage and latency of the clock tree does not change.

4.4.6 Adrenaline for Energy Efficiency

In addition to reducing the tail latency, when needed, Adrenaline can be configured to scale down the voltage and frequency at the query level when the tail latency is much lower than the latency target specified in the service level agreement (SLA). Similar to improving the tail latency, this mechanism works by leveraging query characteristics to identify those queries that are unlikely to have a large impact on tail latency and throttle the voltage during such queries. Adrenaline is flexible enough to adjust policies dynamically based on high-level characteristics of user load, such

as using different latency targets or optimization targets for different levels of load or mixes of query types. Along with the tail reduction approach, we evaluate using Adrenaline to target energy efficiency in Section 5.5.

4.4.7 Responding to Load Changes

The Shortstop circuit can be configured to use different supply voltages on its two voltage rails with a penalty on the order of tens of microseconds. To take advantage of this feature, we employ a load monitor in Adrenaline to monitor the query traffic over time and use this to switch between supply voltage configurations every few seconds to provide maximum benefit to the current query traffic pattern. Note that the fundamental activity of the decision engine is not affected by this tuning, as it continues to make decisions about which queries should be boosted. Since this tuning is done in a much coarser granularity, although it imposes a short pause for changing the voltage on the two voltage rails, it has little impact on the overall distribution of query latency.

4.5 Evaluation

In this section, we evaluate the effectiveness of Adrenaline in both reining the tail latency as well as saving energy. We conduct our evaluations at single-server level, as well as cluster-level for a cluster composed of thousands of servers.

4.5.1 Evaluation Methodology

To accurately evaluate Adrenaline we use the BigHouse simulator [105], which is a datacenter simulation infrastructure that takes workload characteristics to synthesize an event trace to drive a discrete event simulation. We implement Adrenaline and a conventional DVFS baseline in BigHouse. We evaluate both using various configurations.

4.5.1.1 Real system performance characterization

In the BigHouse simulator, two distributions are used to represent a workload: the service distribution and the inter-arrival distribution. We collect these distributions from real system measurements. We use Memcached and Web Search from Cloudsuite [46] as our workloads.

The service time distribution describes how fast the server can process each individual request without counting the queueing latency. To obtain this distribution, we first instrument the Web Search and Memcached server-side software to record the time at the beginning and the end of processing to measure the service time distribution. Then we send requests to the instrumented server one-at-a-time so that no queueing will occur within the server, and collect the latency statistics as the service time distribution.

The machine we use for our service-time distribution measurement features an Intel Xeon CPU E5-2407 v2 @ 2.40GHz CPU, which has 2 sockets with 4 cores per sockets. The size of the main memory is 136 GB. The kernel of the underlying operating system is Linux 3.11.0-15-generic. We carefully deploy the server-side of the benchmarks and the client-side counterpart in a controlled environment to minimize noise due to resource racing and the varying communication over the network, etc. We collect the service-time distribution at each of the frequency steps of the core.

Many workloads have multiple request types, rather than a single type. As prior work suggests [9], there is a big variability in terms of request type composition in production systems and it has a significant impact on the overall performance. Thus we capture this by using the same request type composition as reported in prior work [9] for Memcached, and classifying the requests according to length for Web Search due to the lack of published characterization. Specifically, `APP`, `ETC` and `VAR` refer to the same compositions as described in prior work [9] for Memcached. For Web Search, `ORG` refers to the composition hard-coded in the Web Search client loader, and

SHR, LNG, and UNI refers to synthetic compositions of requests that are short-query-heavy, long-query-heavy, and uniform, respectively. The detailed breakdown of each composition is listed in Table 4.1 and Table 4.2.

We collect the service time distribution for each type of request separately, and evaluate our system under various composition configurations.

Table 4.1: Request type composition of Memcached.

Composition	GET%	SET%	DEL%
APP	83.8	4.7	11.5
ETC	68.6	2.7	28.7
VAR	18.3	81.7	0.0

Table 4.2: Request type composition of Web Search.

Composition	SHORT%	MEDIUM%	LONG%
SHR	53.1	28.7	18.2
LNG	12.7	34.4	52.9
ORG	92.0	5.5	2.5
UNI	34.0	33.0	33.0

Different from service time distribution, inter-arrival distribution heavily depends on the specific configuration and workload. Thus we use the characteristics published by large companies that run these services in production. For Web Search, we follow the guideline from prior work [104], which suggests to use an exponential distribution as an approximation according to the empirical measurement from the production Google Web Search server node. And we use a Generalized Pareto distribution for Memcached as suggested in prior work [9], which characterizes the production Memcached traffic at Facebook.

4.5.1.2 Power modeling

The power equation we are using is adapted based on previous work [101]. We change the exponent term in the equation from 3.0 to 2.7 to consider the effect of imperfect components such as the overhead of the global clock distribution network discussed in Section 4.4.5. The CPU power consumption is described in the following formula:

$$\begin{aligned} P_0 &= P_{sta,0} + P_{dyn,0} \\ P_{sta,0} &= 0.2 \times P_{dyn,0} \\ P_{dyn} &= P_{dyn,0} \times (f/f_0)^{2.7} \\ P_{sta} &= P_{sta,0} \times (f/f_0), \end{aligned} \tag{4.1}$$

where P_0 is the CPU power consumption measured on a real machine, and $P_{sta,0}$ is the static part of P_0 and $P_{dyn,0}$ is the dynamic counterpart; f_0 is the lowest frequency step the machine can run at. P and f are the power and the frequency the CPU is currently operating at.

We instrumented BigHouse to simulate the Adrenaline framework by leveraging multiple service-time distributions in BigHouse on the fly. In addition we model the boosting and decision latencies as well as the requirement to only transition one core from low to high voltage at a time. Upon starting to service a request, our instrumented BigHouse picks the corresponding service-time distribution based on the type of request and the voltage level (always the low-voltage mode at the beginning of a request), and sets the instantaneous power number to be the number that is corresponding to that voltage level. When BigHouse reaches the point of boosting a request, our instrumented BigHouse will calculate the remaining amount of work of a request plus the switching overhead, and use the boosted version of the service-time distribution file to determine the processing time for the rest of the work; the energy

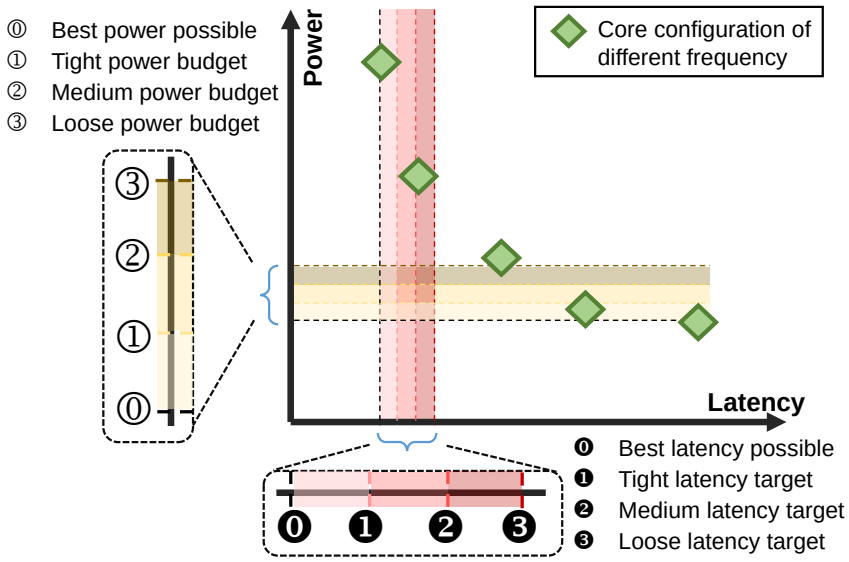


Figure 4.7: Selection of Tight/Medium/Loose power budget and Tight/Medium/Loose latency target for our evaluation.

consumption of the elapsed period of time will be summed up, and the instantaneous power will be updated to the new (boosted) version at the same time. For example, if BigHouse decides to boost from 1.2 GHz to 1.8 GHz at 50% of a request, it calculates the energy used for the first half of the request at the 1.2 GHz power number, and uses the 1.8 GHz service time to process the second half of the request; upon the request completion, it calculates the energy consumption of the second half of the request using the 1.8 GHz power number, and switches back to using the 1.2 GHz service-time distribution for the next incoming request.

4.5.2 Reining in the Tail

In this section, we evaluate the effectiveness of Adrenaline and the conventional coarse-grain sliding window-based DVFS in optimizing the tail latency. Starting from the low frequency as the baseline (e.g. core configuration at bottom right as illustrated in Figure 4.7), which consumes the lowest energy while generating the highest latency, we gradually increase our energy budget and measure the tail latency reduction at both the 95%-tile and 99%-tile.

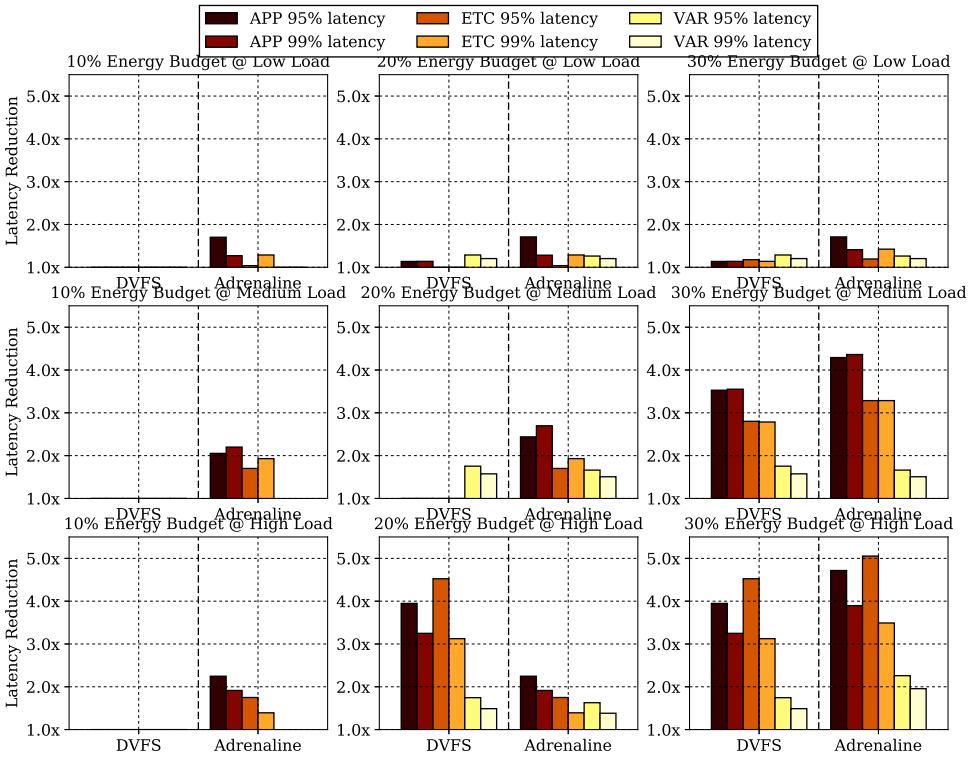


Figure 4.8: Tail latency reduction for Memcached using coarse-grain DVFS vs. Adrenaline. The row presents three load levels and the column represents three energy budgets for boosting. Adrenaline achieves much higher tail latency reduction at various load levels, across workload compositions and energy budgets. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% energy budget.)

Memcached - Figure 4.8 shows the tail latency reduction for Memcached achieved by both coarse-grain DVFS and Adrenaline. In this figure, each row corresponds to a load level (low-, medium- and high-load) and each column corresponds to an energy budget for boosting (low-, medium- and high-budget). The low, medium and high energy budgets are 10%, 20% and 30% of energy increase from the baseline energy, respectively. In each sub-figure, we present the tail reductions achieved by both coarse-grain DVFS (the left cluster of bars) and Adrenaline (the right cluster of bars). Each bar in a cluster represents three real-world workload compositions described in Table 4.1 and the latency reductions for each composition at two percentiles, 95% and 99%. As shown in the first column, given a low energy budget (e.g., 10% energy increase), the coarse-grain DVFS fails to reduce the tail latency, while Adrenaline is able to improve the 95%-tile latency by up to 2.20x and the 99%-tile latency by up to 2.25x. This is due to the fact that the 10% energy budget is too small for the coarse-grain DVFS to boost to the next frequency step, while Adrenaline can take advantage of the fine-granularity nature and boost a small percent of the requests at the tail without consuming much energy. As we increase the energy budgets to 20% and 30%, coarse-grain DVFS starts to show limited improvement on the tail latency. It slightly outperforms Adrenaline for workload `VAR`, because `GET` and `DEL` requests are usually surrounded by groups of `SET` requests in `VAR`. Adrenaline still tries to react to these short-term changes by first switching the core to a lower frequency when the core starts to process these `GET` and `DEL` requests. However, since `GET` and `DEL` requests now have high probabilities to queue behind one or more `SET` requests, they experience long waiting time, and thus become highly likely to be at the tail. Therefore, while Adrenaline does not rein in these `GET`s and `DEL`s promptly, coarse-grain DVFS luckily benefits from its inertia of switching by running at the higher frequency for the entire epoch of requests. However, its improvement on the tail latency is rather limited for the other workload compositions. In general,

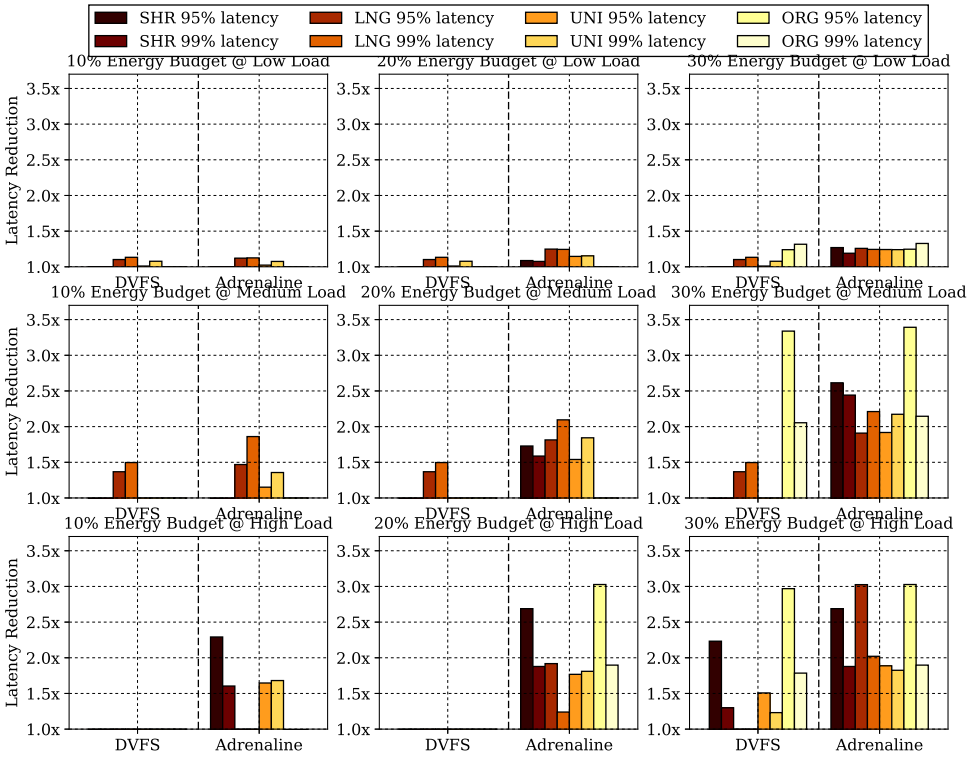


Figure 4.9: Tail latency reduction for Web Search by relaxing energy budget at various load levels using Adrenaline and DVFS. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% energy budget.)

Figure 4.8 shows that, across various request compositions, load levels and energy budgets, Adrenaline almost always improves the tail latency by a larger amount than the coarse-grain DVFS.

Web Search - Similarly, Figure 4.9 presents the tail latency reduction for Web Search at various load levels and across four request composition configurations (specified in Table 4.2). As demonstrated in the sub-figures, Adrenaline shows an edge over coarse-grain DVFS across almost all different workload compositions and energy budgets, especially when the system is at medium and high load levels (the last two rows) where tail reduction is critical. The advantage becomes even more significant when the system is given a higher energy budget. Adrenaline achieves up to a 3.03x tail reduction when given a 20% budget. Given a 30% energy budget, while coarse-grain DVFS starts to take advantage of the large energy head room and show compara-

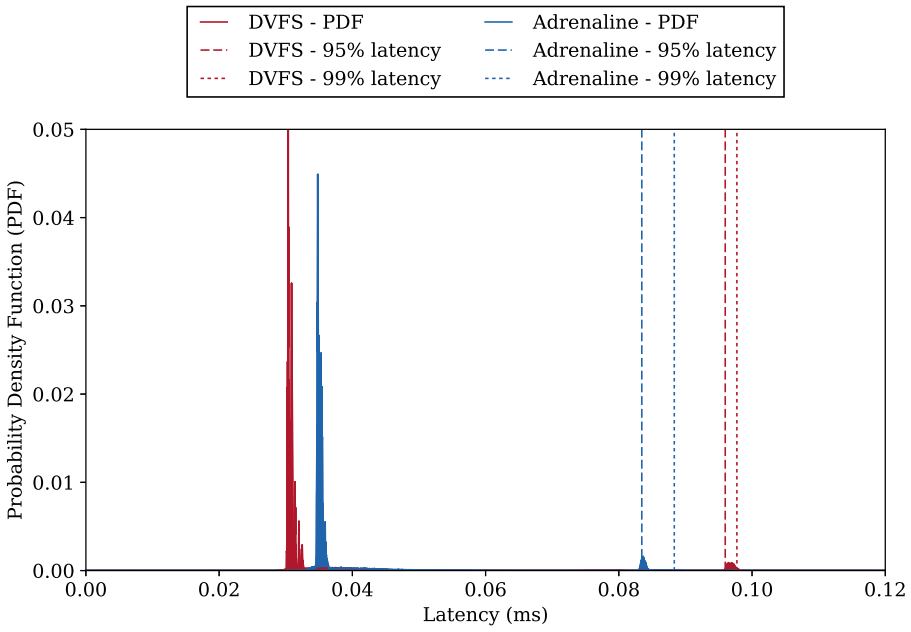


Figure 4.10: Latency probability density function of Memcached when applying DVFS and Adrenaline respectively.

ble tail latency reduction for `ORG`, Adrenaline still achieves better reduction for the same composition, and consistently outperforms coarse-grain DVFS across all other workload compositions.

Latency Distribution - To better understanding the impact of coarse-grain DVFS and Adrenaline on the overall latency distribution, Figure 4.10 compares the distributions achieved by both approaches under the same power budget for boosting. In this figure, the request type composition is `APP` (Table 4.1), composed of 4.7% `SET`, 11.5% `DEL` and 83.8% `GET` requests. This figure presents two probability density functions (PDFs) of the measured request latency of all requests: one measured when using coarse-grain DVFS (red), and the other one using Adrenaline (blue). The dotted lines indicate the 95% and 99%-tile latency of each distribution. For the distributions both coarse-grain DVFS and Adrenaline show `GET` and `DEL` requests are tightly clustered on the left, whereas `SET` requests have a much higher latency and compose the long tail.

Compared to the no-boosting scenario, the coarse-grain DVFS shifts the entire

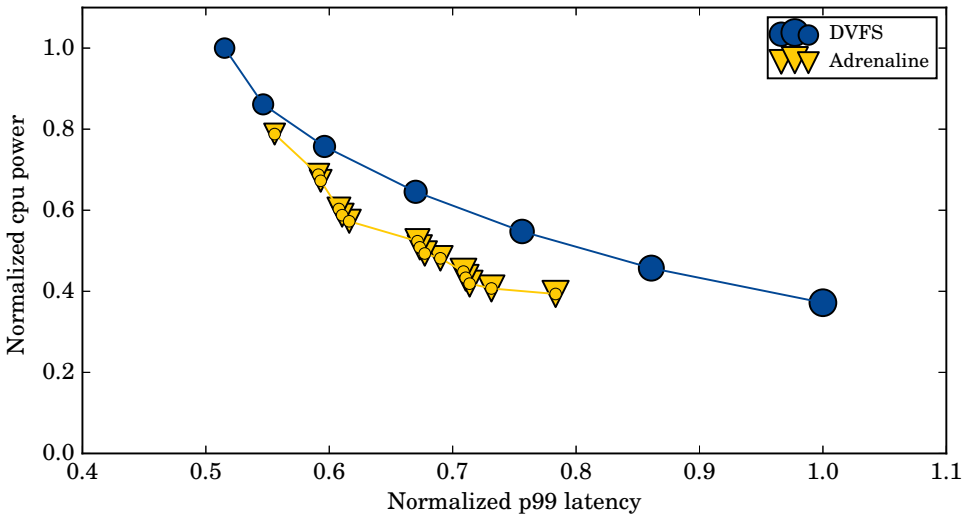


Figure 4.11: The scatter plot of Memcached, in which each data point represents the normalized power/latency performance of a single CPU V/f configuration with load composition `APP` and low traffic.

distribution to the left, reducing both the mean and 99%-tile latency by a small amount. However, Adrenaline spends the limited power budget for boosting in a more effective way. Instead of boosting all requests including both the fast requests and the tail requests, it only boosts the requests that have high probability to be in the tail. The mean latency Adrenaline achieves is slightly slower than that achieved by the coarse-grain DVFS, but it is still faster than the no-boosting scenario. More importantly, Adrenaline achieves a much more significant reduction on the tail latency than the coarse-grain DVFS.

Figure 4.11 shows a Pareto-optimal curve for the power versus 99% latency tradeoff of different V/f configurations for both DVFS and Adrenaline. At extremely tight SLA targets the DVFS approach is necessary, but as soon as the latency target is loosened the Adrenaline approach offers either significant reductions in tail latency (further to the left at a given power budget) or a gain in power efficiency (lower in the graph for a given target latency).

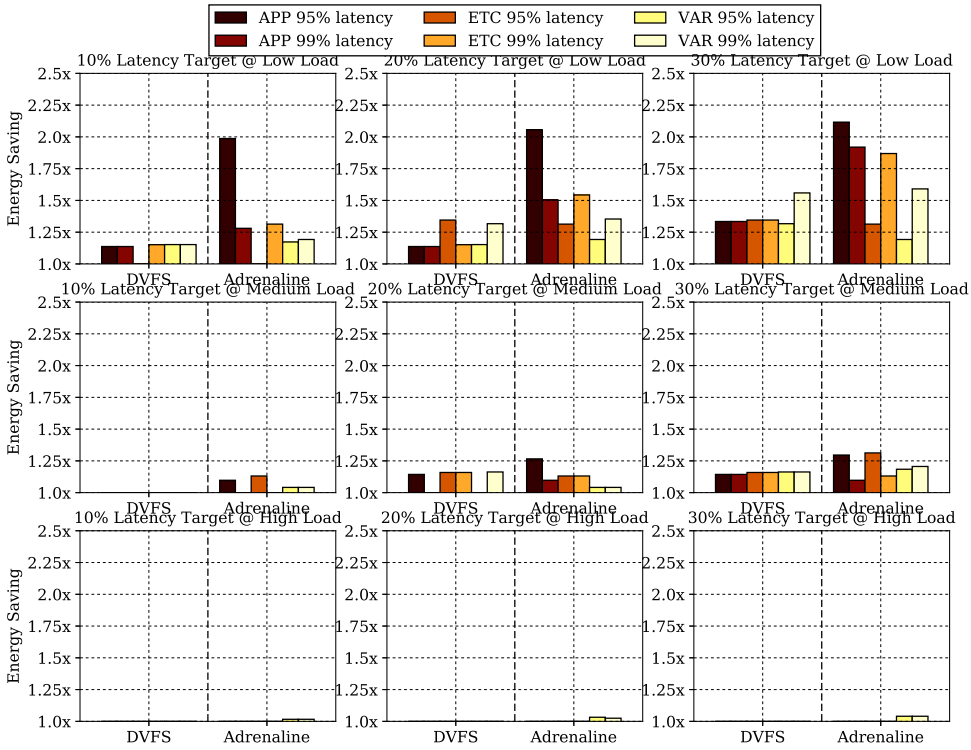


Figure 4.12: Energy saving for Memcached by relaxing the tail latency target at various load levels using Adrenaline and coarse-grain DVFS. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% latency budget.)

4.5.3 Energy Saving

In addition to reining the tail latency by fine-grain boosting, Adrenaline can also be used to take advantage of the latency slack, especially at low load, to improve energy efficiency. As illustrated in Figure 4.7, we start at the highest CPU frequency, which generates the lowest possible latency, and evaluate the effectiveness of coarse-grain DVFS and Adrenaline in improving the energy efficiency as we gradually relax the tail latency target. Specifically, we refer 10% latency lack as the strict target, 20% as the moderate target and 30% as the relaxed target in our experiments.

Figure 4.12 presents the energy savings for Memcached achieved by coarse-grain DVFS and Adrenaline, respectively. As demonstrated in the figure, given the same latency target, Adrenaline can significantly reduce the energy consumption, especially at low load. It achieves up to a 2.12x energy savings over the no-DVFS baseline,

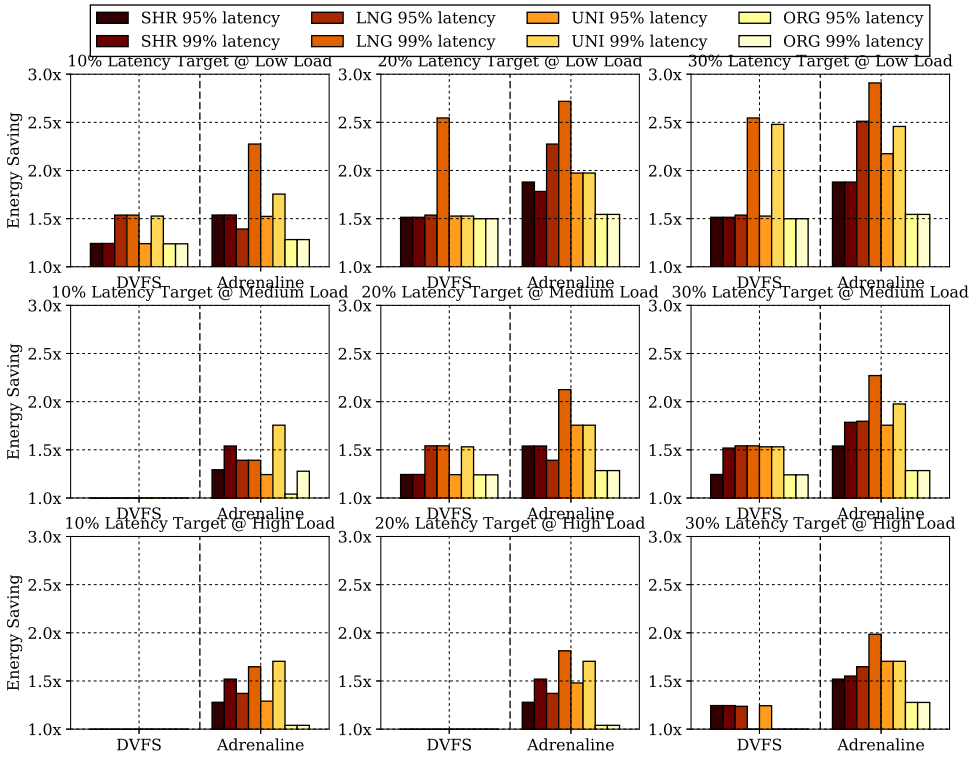


Figure 4.13: Energy saving for Web Search by relaxing the tail latency target at various load levels using Adrenaline and DVFS. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% latency budget.)

whereas DVFS can only save 1.56x in the best scenario. At high load, both Adrenaline and coarse-grain DVFS cannot achieve much energy savings. This is to be expected because at the high load, there is not much latency slack due to the queuing delay. It is very hard to achieve energy savings unless we are willing to sacrifice a great amount of tail latency slack. Similarly, Figure 4.13 shows that Adrenaline can achieve significant energy savings across all three different load levels, especially at low and medium load levels, for Web Search. In addition, Adrenaline often achieves greater energy savings compared to coarse-grain DVFS. This is because Adrenaline leverages the observation demonstrated in Figure 4.1 and Figure 4.2, and prioritizes the boosting of those requests which give the most benefits, which effectively prevents Adrenaline from wasting energy on those requests with low boostability.

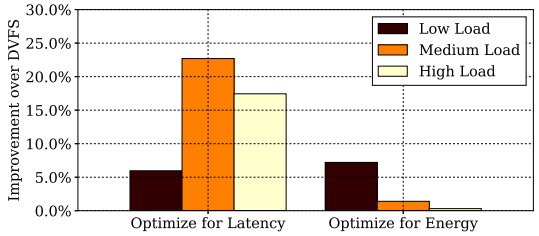


Figure 4.14: Improvement of Adrenaline over coarse-grain DVFS for Memcached.

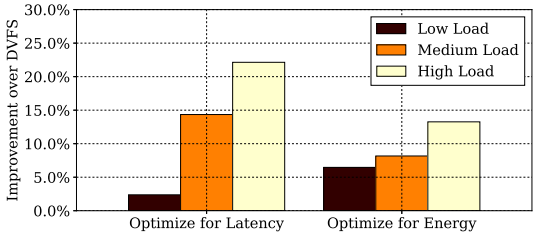


Figure 4.15: Improvement of Adrenaline over coarse-grain DVFS for Web Search.

4.5.4 Overall Comparison

Figures 4.14 and 4.15 present Adrenaline's improvement of tail latency and energy savings over DVFS, averaged across all workload compositions. Overall, Adrenaline outperforms coarse-grain DVFS in almost all the situations. When optimizing for tail latency, Adrenaline achieves up to 22.7% improvement over DVFS by dynamically scaling the frequency at a finer granularity to target only the most important requests.

4.5.5 The impact of boost threshold

We analyzed the time non-candidate queries spent in the system and use a **boost threshold** to predict long-running queries that should be boosted. This boost threshold is the product of the QoS target and a coefficient, as shown in Equation 4.2:

$$T_{threshold} = c_t \times T_{QoS}. \tag{4.2}$$

For example, for the previous studies in Section 4.5.2 and Section 4.5.3, our choice of boost threshold was 0.5x of the QoS target. This choice translates into a threshold coefficient c_t of 0.5.

While our results shown in the previous sections show that we achieve significant improvement in both tail latency reduction and energy saving with a straightforward selection of $c_t = 0.5$, we observe that the choice of c_t can have significant impacts on performance and efficiency of Adrenaline for benchmarks with different characteris-

tics. To understand the effects, we evaluate Adrenaline under a range of threshold coefficient settings, and study their impact on the reduction of the 95th percentile latency given certain energy consumption constraints. Note that the coefficient c_t determines only the boost threshold for non-candidate types of queries (i.e., `GET` and `DEL` for Memcached and `MEDIUM` and `LONG` for Web Search), whereas the candidate types of queries (i.e., `SET` for Memcached and `SHORT` for Web Search) will always be boosted at the onset of core processing.

4.5.5.1 A general principle

The goal of Adrenaline is to rein in the tail queries to meet the QoS target. To accomplish this, it is desirable for the boost threshold to be smaller than the QoS target. This allows tail queries to be identified before they miss the target. But one challenge here is that rather than ignoring those queries that are fast enough, when the threshold is too small, Adrenaline boosts most of the queries. This failure to consider query characteristics leads to energy waste.

4.5.5.2 Memcached vs. Threshold Coefficient

To study the impact of boost threshold on the performance of Memcached, we sweep c_t within the range of 0.0001 to 0.5, which translates to a range from 1 microsecond to 5 milliseconds for the boost threshold $T_{threshold}$. Figure 4.16 shows the effect of choosing different c_t on tail latency reduction for Memcached, given certain energy budgets. We prefetch and conclude our major findings as follows:

- **Finding 1** Adrenaline's performance is sensitive to the choice of c_t. The performance variation, with the same workload, in tail latency reduction and energy saving can be as large as 5x and 1.78x, respectively.

- **Finding 2** Adrenaline configured with optimal c_t can extract up to 4.82x improvement over coarse-grain DVFS in the reduction of the 95th-percentile la-

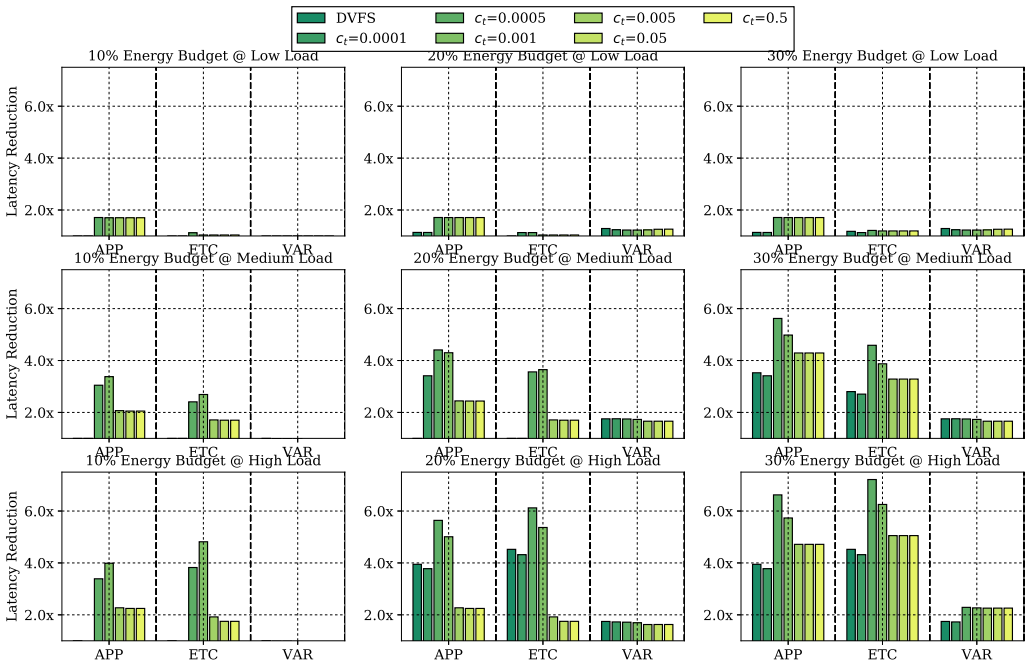


Figure 4.16: Sensitivity analysis of the effect of different boost thresholds on tail latency reduction of Memcached. Row 1 to 3 are the results of sensitivity analysis with low, medium, and high load, respectively. Column 1 to 3 are the results with 10%, 20%, and 30% energy budget, respectively.

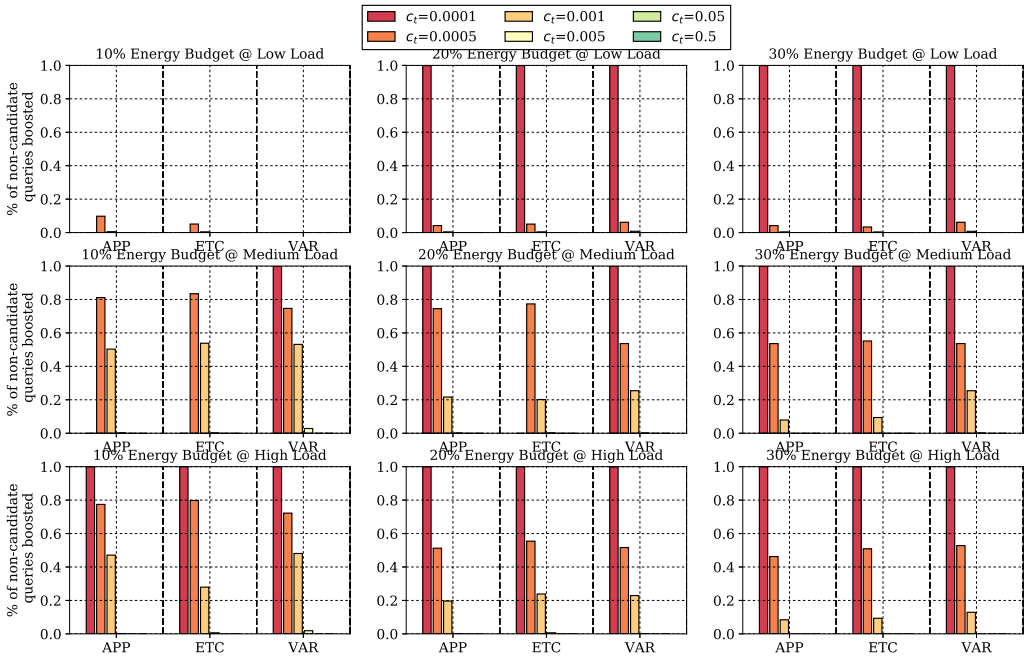


Figure 4.17: Percentage of non-candidate Memcached queries (`GET`s and `DEL`s) boosted when optimizing for tail latency. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% energy budget.)

tency, even under the tightest energy budget.

- **Finding 3** The optimal choice of c_t is within the range of 0.0005 and 0.001. This optimal choice range remains almost constant across different loads and energy/latency budgets.

Choosing an optimal c_t that minimizes the tail latency while remaining within the energy budget can be challenging. Too low a threshold wastes considerable amounts of energy to boost queries that are natively fast enough. Too high a threshold fails to effectively reduce query latency due to the limited percentage of the execution that is boosted.

To explain the importance of a proper threshold, we refer to both Figure 4.16 and Figure 4.17. Figure 4.16 shows Adrenaline's impact on tail latency reduction with different choices of c_t across different load levels and energy budgets. Figure 4.17 gives the percentage of boosted non-candidate queries. When c_t is large, Adrenaline boosts very few or no non-candidate queries, generating little positive effect on tail latency. As we decrease c_t, Adrenaline boosts an increasing amount of non-candidate queries and therefore yields better performance. When $c_t = 0.0005$ and $c_t = 0.001$, Adrenaline achieves the optimal improvement across different load levels and energy budgets. As can be seen in Figure 4.17, Adrenaline properly boosts a large portion, but not all, of the queries. This indicates that the small threshold coefficients allow Adrenaline to boost queries early enough and therefore reduce the probability of forming a long queue, resulting in shorter queueing delays for queries in the queue.

This outcome is highly correlated to the characteristics of different request types of Memcached shown in Figure 4.1. While the service times of all types of Memcached queries are generally short and the coefficients of variation are small, there is a clear distinction between the latency distribution of the candidate type (`SET`) and the distributions of the two non-candidate types (`GET` and `DEL`). If we set c_t to 0.0005 or 0.001, Adrenaline can save energy by using the lower voltage to serve the queries

when the queue is short and is composed of mostly non-candidate queries; when the queueing delay becomes critical, Adrenaline can agilely enter the boost mode. If we further lower c_t to 0.0001, however, the benefit on tail latency reduction disappears. We can conclude from Figure 4.17 that, Adrenaline with $c_t = 0.0001$ boosts almost 100% of the queries. Under this condition, the energy overhead incurred by running the majority queries with high-performance configuration is simply too high for the energy budget to afford. The choices of supply voltage on the two voltage rails in Shortstop are, therefore, restricted to the lower ones, which limits the speedup Adrenaline can achieve.

Figure 4.17 reveals some cases where the result of $c_t = 0.0001$ is missing. For example, the results of $c_t = 0.0001$ of all the three query compositions with 10% energy budget at low load, of `APP` and `ETC` with 10% energy budget at medium load, and of ETC with 20% budget at medium load are absent. The absence of these results indicates that the low threshold makes Adrenaline boost queries too early, largely increasing the overall energy consumption, making Adrenaline violate the given energy budget no matter which CPU V/f configuration is used.

Figure 4.18 and Figure 4.19 show boost threshold's impact on energy saving. We see from these two figures that, when the system is under medium and high load, there is a trend similar to what we have seen in the previous experiments. However, one counter-intuitive finding for workload `ETC` at low load is that, as we decrease c_t from 0.001 to 0.0005 and Adrenaline is given a 20% latency target, the energy saving becomes more significant even though the percentage of boosted non-candidate queries increases. This is due to the fact that Adrenaline is using a different of supply voltages for the Shortstop voltage rails. When $c_t = 0.001$, because the threshold is high and the workload `ETC` contains only a very small percentage of `SET` queries, only a few queries get boosted; under this condition, Adrenaline is not able to meet the 20% tail latency budget without using the highest V/f setting for its boost mode.

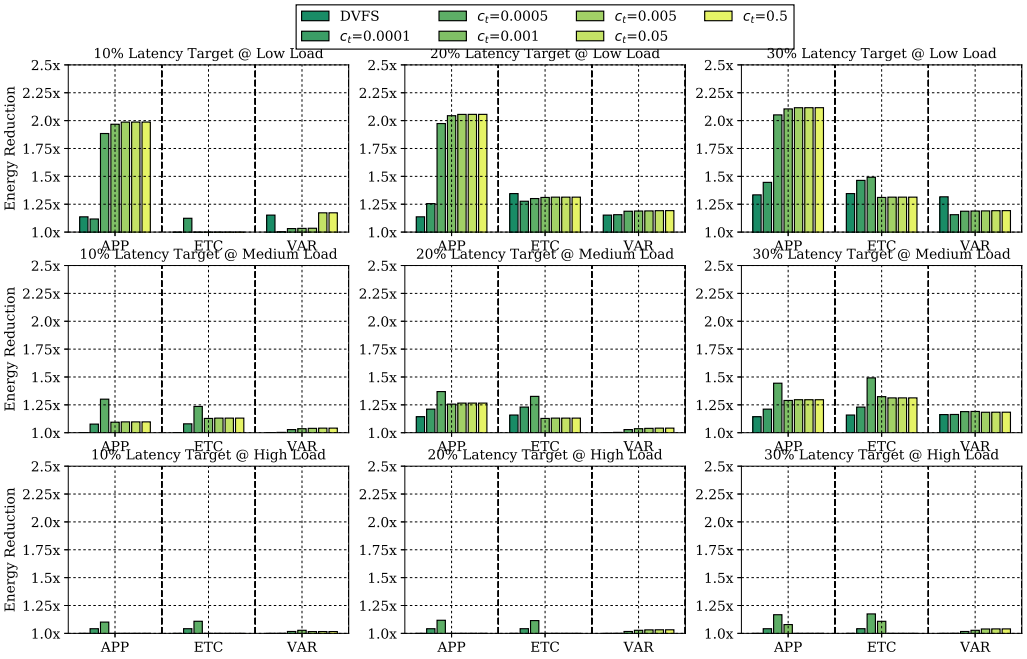


Figure 4.18: Sensitivity analysis of the effect of different boost thresholds on energy reduction of Memcached. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% latency budget.)

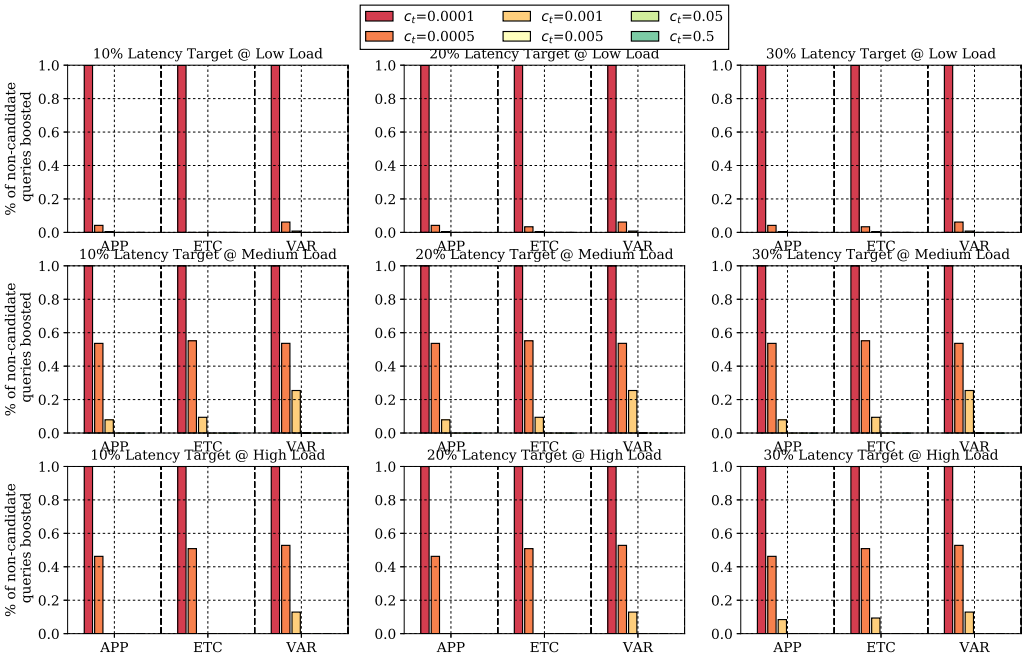


Figure 4.19: Percentage of non-candidate Memcached queries (`GET`s and `DEL`s) boosted when optimizing for energy saving. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% latency budget.)

But once we decrease c_t to 0.0005, Adrenaline enters boost mode much earlier, which mitigates the queueing effect; a V/f configuration with a lower boost-mode frequency (i.e., the frequency used when Shortstop is supplied by its high-voltage rail) is now sufficient for Adrenaline to meet the tail latency target.

We also see from the two figures that, larger energy budgets sometimes lead to fewer boostings compared to smaller energy budgets at the same load. This is because the larger energy budget gives Adrenaline the flexibility to use a more aggressive V/f configuration which enables a higher normal-mode frequency (i.e., the frequency used when Shortstop is supplied by its low-voltage rail). This higher normal-mode frequency benefits all of the queries in the incoming query stream and shortens their latency, leading to fewer needs for boostings.

4.5.5.3 Web Search vs. Threshold Coefficient

We conduct a similar sensitivity analysis for Web Search. For Web Search, we sweep the threshold coefficient c_t from 0.001 to 0.5, which translates to a range from 50 microseconds to 250 milliseconds for the boost threshold $T_{threshold}$. We again prefetch our findings and present them as follows:

- **Finding 1:** For Web Search, Adrenaline's impact on latency reduction and energy saving is also sensitive to the choice of c_t. For a same workload, the performance variation across different choices of c_t can be up to 3.17x in tail latency reduction and up to 1.62x in energy saving.

- **Finding 2:** The optimal choice of c_t for Web Search is within the range of 0.1 to 0.5, which is much larger than that for Memcached. This distinction results from the different characteristics the Web Search queries have. However, the optimal choice still remains almost constant across different load and different energy/latency budgets.

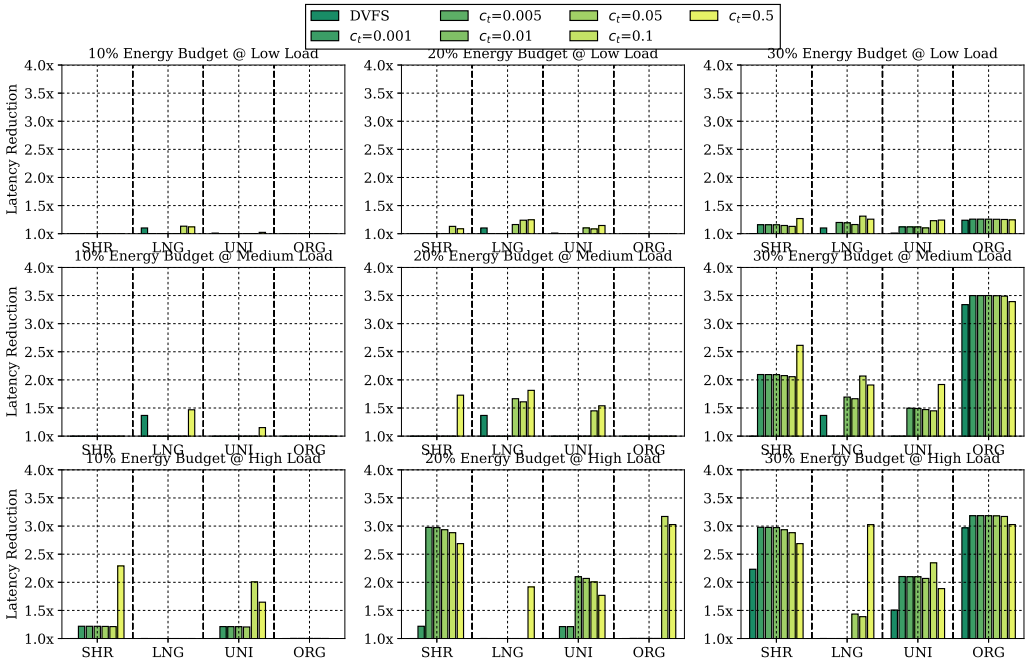


Figure 4.20: Sensitivity analysis of the effect of different boost thresholds on tail latency reduction of Web Search. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% energy budget.)

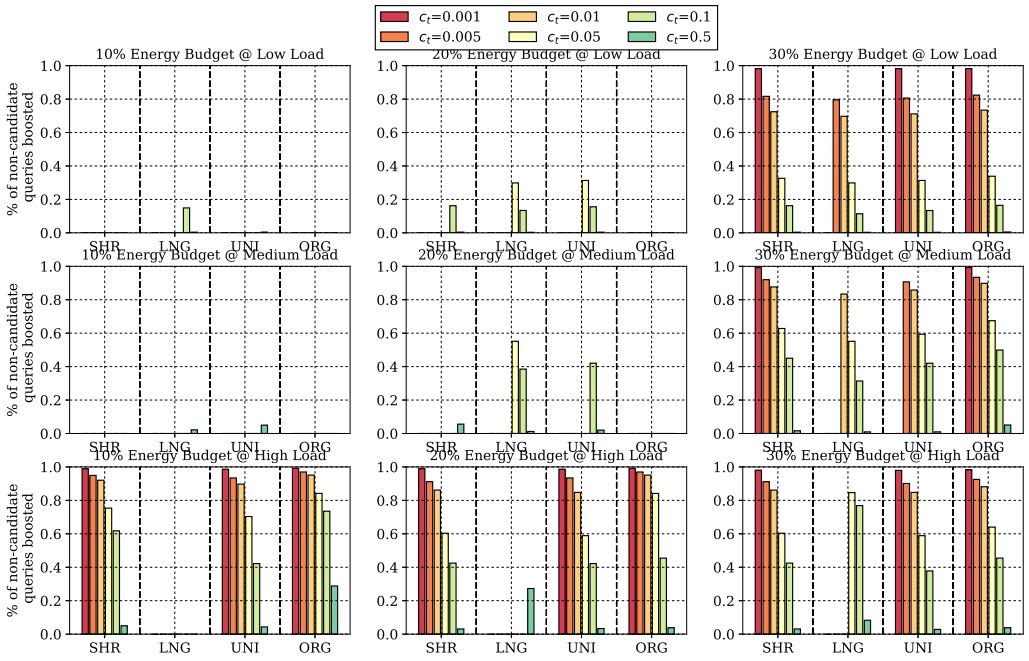


Figure 4.21: Percentage of non-candidate Web Search queries (`MEDIUM`s and `LONG`s) boosted when optimizing for tail latency. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% energy budget.)

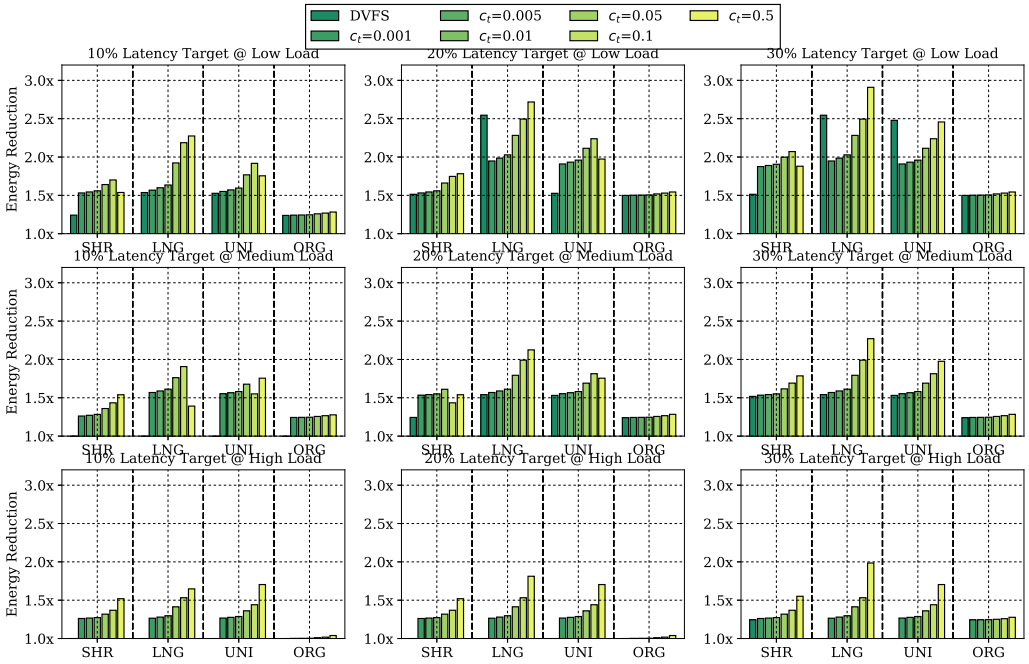


Figure 4.22: Sensitivity analysis of the effect of different boost thresholds on energy saving of Web Search. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% latency budget.)

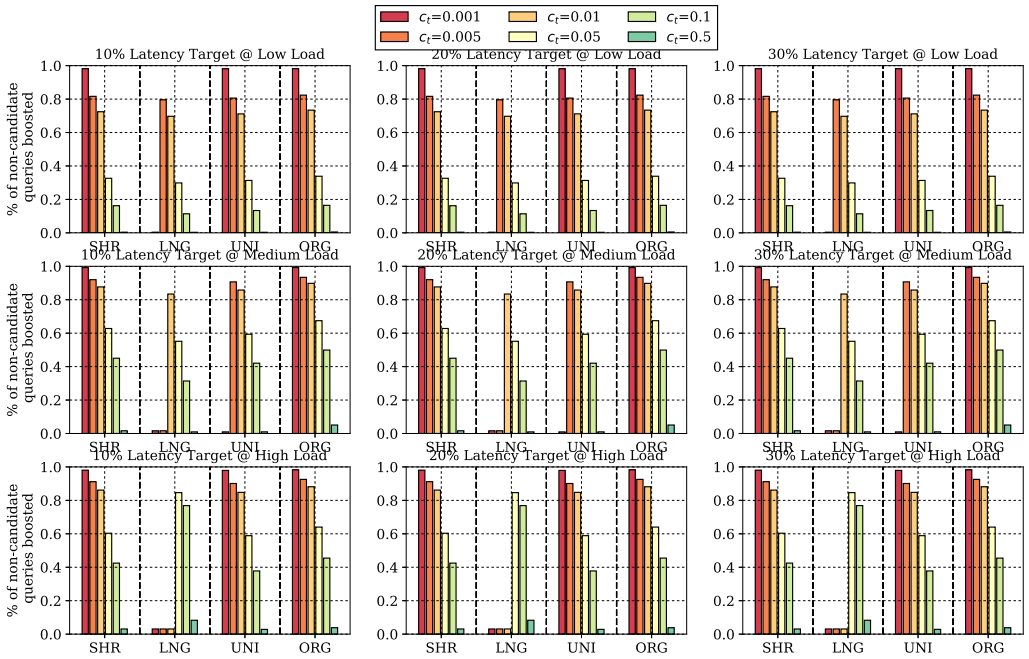


Figure 4.23: Percentage of non-candidate Web Search queries (`MEDIUM`s and `LONG`s) boosted when optimizing for energy saving. (Row 1 to 3 – low, medium, and high load. Column 1 to 3 – 10%, 20%, and 30% latency budget.)

As can be seen in Figure 4.20, Adrenaline, again, provides significant improvement over coarse-grain DVFS for Web Search at medium load and high load. When we study Adrenaline's effectiveness in tail latency reduction across the different choices of c_t, however, we find that the system behavior is drastically different from what we've seen in the Memcached experiments. As shown in Figure 4.20, the larger c_t's such as 0.1 and 0.5 now have a better chance to provide significant tail latency reduction over coarse-grain DVFS. The optimal c_t for Web Search is higher compared to Memcached, because Web Search service queries usually need a much longer time to process, as shown in Figure 4.2. The non-candidate Web Search queries, and especially the `LONG` queries, are very insensitive to the change of the frequency boosting. This means that when Adrenaline enters the boost mode too early when processing non-candidate queries, it wastes significant amounts of energy but only yields little latency reduction. So c_t should be set to a higher value so that the cores stay at the normal mode, and boost only when urgent conditions occur. Similar findings are made when we use Adrenaline to optimize for energy saving. Figure 4.22 and Figure 4.23 show that, under the same load level and the same tail latency target, Adrenaline achieves more significant energy saving when using larger c_t's.

Seeing this significant improvement, one might find it surprising in Figure 4.21 that the boost percentages of higher c_t's are low. The reason why such low boost percentages can lead to significant tail latency reduction is that Adrenaline is using higher voltage values for Shortstop's high-voltage rail. With these higher boost-mode voltages, Adrenaline is able to use higher boost-mode frequencies to effectively shorten the latency of candidate queries and some of the long-running non-candidate queries, which compensates for the low boost percentages. Due to the low boost percentage, even though Adrenaline uses a higher boost-mode voltages, the energy budget constraints are not easily violated.

4.5.5.4 Strategy for choosing the right boost thresholds and V/f configurations

In this section, we discuss the strategy for choosing the right boost thresholds and voltage configurations at runtime when deploying Adrenaline for an application.

During runtime, Adrenaline makes two types of decisions for different purposes. The first type of decision is the per-query boost decision. This type of decisions, as we discussed in the previous sections of the chapter, aims to determine whether to boost processor performance for each query. Since user-facing applications have strict latency requirement and have an enormous amount of user queries hitting the servers every second, the fine-grain decision engine needs to be low-overhead. Any significant extra latency added by this engine builds up non-negligible delays and slow down all the queries drastically in a loaded system. Adrenaline achieves low overhead by looking for simple query-level indicators to make fast decisions which switch the underlying Shortstop circuit between its high/low voltage rails, incurring only nano-second-level delays.

The second type of decisions reacts to the change of traffic level and query composition, determining the voltage configuration of the underlying Shortstop circuit and the boost threshold used for accelerating non-candidate queries. The decision engine needs to collect enough information over a sliding window before it can explore a suitable next configuration to adapt to. In addition, changing the voltage configuration incurs significant overhead; instead of switching quickly between the high and low voltage rails, executing this type of decision involves adapting the voltages on the two voltage rails via an off-chip regulator. The delay of such an operation is within the range of tens of microseconds [42], easily comparable or even exceeding the average service time of shorter queries, such as `GET`, `SET`, `DEL` queries in Memcached. This type of decisions, therefore, should only be triggered at a coarser time granularity.

However, it is challenging to develop a generalized decision-making runtime algo-

rithm suitable for multiple applications. As shown in Section 4.5.5, the optimal boost threshold and the optimal V/f configuration of Adrenaline vary significantly under different situations. Since the decision highly depends on the characteristics of queries of the target application, as well as the current traffic level and query composition, the construction of such a generalized algorithm might involve designing complicated control-theoretic mechanisms, which is out of the scope of this work.

In this work, we advice to construct such a decision engine on top of profiling results of the target application, which can be done as discussed in Section 4.5.5.2 and Section 4.5.5.3. We provide some guideline as follows:

Strategy for choosing boost threshold. From the results shown in the previous section, we observe that boost threshold is highly related to the target application's typical service time, typical query arrival rate (in terms of query-per-second, QPS), and non-candidate queries' responsiveness to core frequency. Boost threshold is rather insensitive to the change in query composition and traffic level. As shown in Figure 4.16 and Figure 4.22, the optimal threshold coefficients for an application is almost always the same under different combinations of query compositions and traffic levels. For Memcached, $c_t = 0.0005$ or $c_t = 0.001$ is always the choice for threshold coefficient. For Web Search, $c_t = 0.5$ or $c_t = 0.1$ is almost always superior than other c_t's; in the cases where neither of them is the best, their performance still comes close to the optimal ones. Such a characteristic greatly simplifies the strategy for choosing a suitable boost threshold at runtime. For example, for applications similar to Memcached, which feature short query service time, high QPS, and non-candidate queries that are highly core-frequency-responsive, boost decisions should be made early enough to prevent long queues from building up. In this situation, smaller c_t's should therefore be prioritized over larger c_t's. On the other hand, applications similar to Web Search have long query service time, much lower QPS, and unresponsive non-candidate queries. Adrenaline takes advantage of this characteristic and use a

larger threshold which boosts non-candidate queries later, gaining significant energy saving without building up a long queue.

Strategy for choosing V/f configuration. After we conduct an off-line profiling on the target application, for each combination of traffic level and query composition, we identify the most beneficial V/f configuration, and record this information in a lookup table. During runtime, we use a sliding window to monitor the most recent history of workload. According to workload observed in the window, the decision engine chooses the next configuration by searching for the V/f configuration associated with the closest combination of traffic level and query composition in the lookup table.

For user-facing applications, since their traffic usually follow a diurnal pattern [104] and are rather stable within a short period of time, meaning that the choice for the next decision is more predictable and reliable if given the recent history of traffic. For these applications, datacenter operators can quantize traffic and query compositions into many levels, and create a fine-grained lookup table, which provides more precise V/f decisions. On the other hand, if the target application features a varying and unpredictable traffic pattern, datacenter operators can use a coarser-grain lookup table to prevent the decision engine from overfitting to sudden bursts of traffic.

4.5.6 Tail at Scale

Many large-scale web services including Memcached and Web Search use one or even many clusters of machines to serve user queries. As presented in prior work [12, 111], such web services tend to have a large amount of inter-communications and high fan-out. For example, a single user `HTTP` request can result in hundreds of Memcached data fetches and many round trips within a cluster at Facebook. As the number of servers involved in serving a user request increases, the probability of violating the Quality of Service target (e.g., request latency target) grows significantly [32].

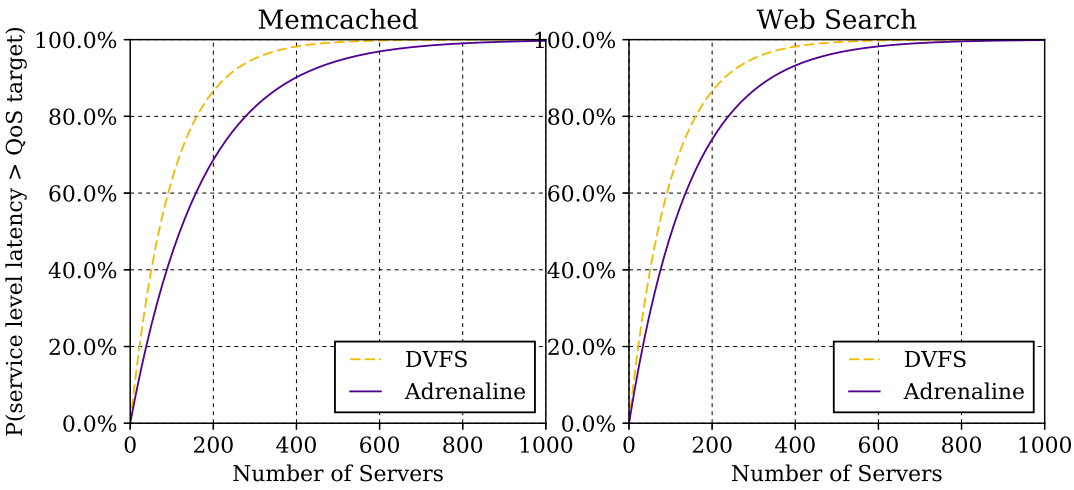


Figure 4.24: Effectiveness of Adrenaline in reducing the tail latency at service level comparing to DVFS.

In this section, we evaluate the effectiveness of Adrenaline in reducing the service-level tail latency violations by reducing the tail latency at each leaf node. We use the 99% tail latency measured when the coarse-grain DVFS is enabled as the service-level latency target, and compare the probabilities of one request in a fan-out cluster violating the service-level QoS target when using coarse-grain DVFS versus Adrenaline. Figure 4.24 presents the probability that at least one leaf node misses its QoS target as a function of the number of leaf nodes for a service that must wait for all leaves to respond. As shown, the probability of a request violating the service level QoS target increases drastically as the number of servers in the fan-out cluster increases. With 100 servers, the probability of one request violating the service-level target has increased to 63% for the coarse-grain DVFS. By enabling Adrenaline to reduce the tail latency at leaf nodes, we are able to reduce the probability with 100 servers significantly, by 19% for Memcached and by 11% for Web Search. With 200 servers, Adrenaline achieves an 18% improvement over DVFS for Memcached and a 13% improvement for Web Search.

4.6 Summary

In this chapter, we present the Adrenaline methodology that adjusts the voltage/frequency at query-level granularity to rein in the tail latency as well as to save energy. By evaluating our methodology under various realistic workload configurations, we demonstrate the effectiveness of our methodology. With a naive boost threshold for non-candidate queries, we achieve up to a 2.50x tail latency improvement for Memcached and up to a 3.03x for Web Search over coarse-grain DVFS given a fixed boosting power budget. When optimizing for energy reduction, Adrenaline achieves up to a 1.81x improvement for Memcached and up to a 1.99x improvement for Web Search over DVFS. By using the carefully chosen boost thresholds, Adrenaline further improves the tail latency reduction to 4.82x over coarse-grain DVFS.

CHAPTER V

Slingshot: Fine-grain Resource Scheduling for Reconfigurable Datacenter Hardware

This chapter outlines the motivation and design of a novel CPU-FPGA system to achieve high power efficiency and low latency for the highly dynamic cloud environment. Prior works [121, 21] have reported successes in using FPGA to accelerate cloud workload in large-scale datacenters; however, the FPGAs deployed in prior works are statically configured during runtime, leading to limited utilization, and need to be taken offline for reconfiguration when workload changes. To fully exploit the potential of the state-of-the-art CPU-FPGA system, in this chapter, we introduce Slingshot: a combined workload scheduler and FPGA resource manager to enable FPGA resource sharing. We begin by characterizing two state-of-the-art CPU-FPGA platforms: Intel's Heterogeneous Architecture Research Platform (HARP) that provides low-latency communication over Intel's QuickPath Interconnect (QPI), and a high bandwidth solution connecting the CPU and FPGA over PCIe. We use these real-system measurements to design a datacenter model to study FPGAs as a shared service. We design scheduling policies to maximize load capacity while minimizing latency when dispatching queries to CPU and FPGA accelerator units and dynamically repartitioning the FPGA. Slingshot achieves a 96% prediction accuracy in assigning queries to the optimal resource. In addition, Slingshot can either improve the 99th

percentile tail latency by 1.7×-8× over statically-configured systems, or increase the server load capacity by 23-70% while maintaining the same 99th percentile tail latency as statically configured systems.

5.1 Goal of Study

The aspects of FPGA resource sharing are particularly challenging to study in the dynamic datacenter environment. Compared to other general-purpose platforms such as CPUs and GPUs, resource sharing for FPGAs has a much larger design space due to the complex heterogeneity in resource requirement among different accelerators, and there has not been an explicit mechanism to handle this type of resource sharing.

In this chapter, we aim to address this limitation by proposing a resource sharing system specifically for FPGAs. We present the design of Slingshot, a robust mechanism to share FPGA resources spatially and temporally across multiple dynamic workloads. Using Slingshot, queries from different applications can be flexibly served throughout the datacenter without being restricted by the present configuration of FPGA resources, and each machine dynamically decides whether to process queries on the CPU or FPGA using resource-aware scheduling algorithms. The end result is that Slingshot effectively treats the FPGA as a shared service.

Using Slingshot's FPGA-as-a-service model, each server dynamically reprograms its own FPGA on the fly to best cope with changing load patterns (e.g., increasing the available accelerators of a particular type in response to a sudden spike in relative load). In this chapter, we present a datacenter model we construct based on the real-system measurements and study the impact of different partitioning and scheduling policies for dynamically reconfigurable FPGA acceleration.

We design a robust methodology to share FPGA resources across multiple dynamic workloads. Using the proposed system, queries from different applications can be served throughout the datacenter with regard to the present configuration of FPGA

resources, and each machine dynamically decides whether to process queries on CPU or on FPGA based on resource-aware scheduling algorithms, effectively treating the FPGA as a shared service. Using such an FPGA-as-a-service model, each server can dynamically reprogram the FPGA to better cope with rapidly changing load patterns (e.g., increasing the logic elements allocated an accelerator in response to increasing load). Specifically, we make the following contributions:

- **In-Depth Characterization of Resource Sharing on FPGAs -** We characterize two FPGA server systems to understand how applications can share hardware resources both spatially and temporally. One system connects the CPU to FPGA via PCIe, and the other is Intel's HARP prototype, which connects the CPU and FPGA via a memory-coherent QPI bus (Section 5.2).

- **Datacenter Architecture -** We present a datacenter architecture design with FPGAs integrated at the server level, and discuss the resource allocation and job scheduling considerations that must be addressed to efficiently utilize FPGAs in a dynamic datacenter environment (Section 5.3).

- **Scheduling Algorithm Quantitative Study -** We present Slingshot, a set of resource-aware algorithms to dynamically allocate work between the host CPU and FPGA accelerator and choose when to reprogram the FPGA fabric at runtime for the best throughput and latency (Section 5.4).

- **System-level Results -** Using the performance measurements from real hardware platforms, we evaluate Slingshot's effectiveness in a datacenter environment and show improvements in colocated workloads. We show that Slingshot can be used to either increase the Quality of Service (QoS) of multiple workloads colocated on a CPU-FPGA system over a statically programmed system, or increase the maximum load capacity of over a static system (Section 5.5).

5.2 System and Workload Characterization

Prior works statically provision FPGAs in datacenters to accelerate single workloads, which heavily underutilizes the FPGA resource as illustrated in Figure 1.3. To investigate whether and how we can leverage FPGAs as a shared resource among applications, there are several critical questions we need to answer.

- How are modern CPU-FPGA systems architected?
- How can the FPGA fabric be shared?
- How do real-world datacenter workloads perform on modern CPU-FPGA systems?

5.2.1 CPU-FPGA System Architecture

To answer these questions, we characterize two state-of-the-art CPU-FPGA systems: a memory-coherent CPU-FPGA system connect via QPI (Intel's Heterogeneous Architecture Research Platform (HARP)), and a CPU-FPGA system connected via PCIe. The full system specifications are outlined in Table 5.1.

Table 5.1: System specifications

	HARP System	**Stratix System**
CPU	Xeon E5-2680v2	Xeon E5-2680v2
CPU Freq.	2.80 GHz	2.80 GHz
Cores	10	10
FPGA	Stratix V	Stratix V
FPGA Freq.	200 MHz	200 MHz
Interconnect	QPI	PCIe

5.2.2 Interconnect Sharing

To enable resource sharing on FPGAs, the interconnect needs to be shared among applications. Therefore, we characterize the performance of the interconnect on the two CPU-FPGA systems in this section.

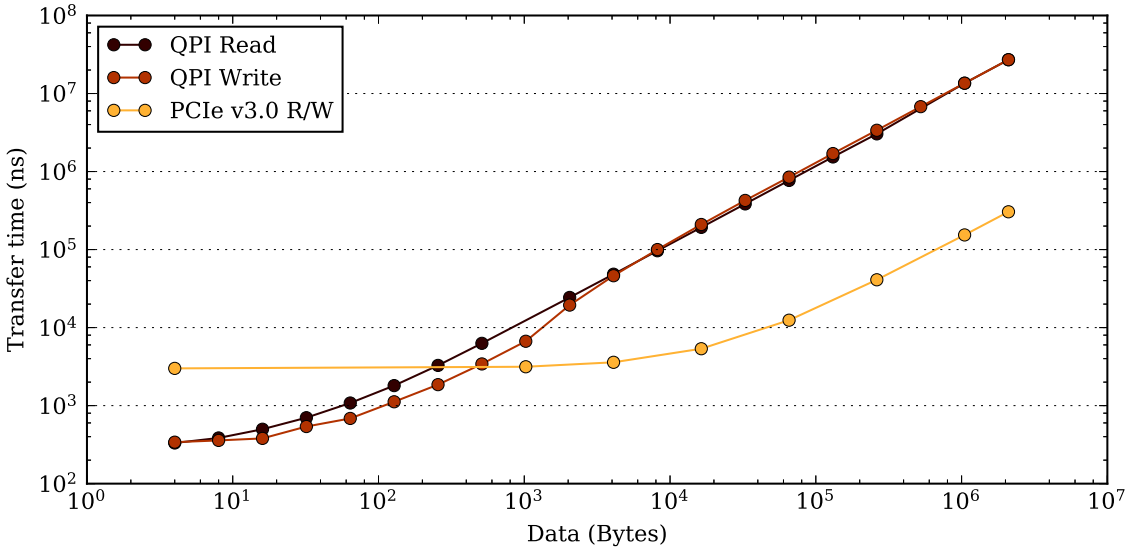


Figure 5.1: Measured data transfer time from the CPU to FPGA (lower is better). PCIe has dedicated upstream/ downstream channels whereas QPI is bidirectional.

In Figure 5.1, we plot the measured data transfer time from the CPU to the FPGA for different packet sizes. For both QPI and PCIe, we measured the total transfer time from main memory (CPU DRAM) to the FPGA's on-chip cache. Each datapoint in Figure 5.1 represents the average of 1000 samples.

From our characterization, we measured QPI's peak read speed at 5.5 GByte/s and peak write speed at 4.9 GB/s. We measure the minimum memory latency at 310 ns. For PCIe 3.0, we measure the peak read and write speeds both at 6.8 GB/s and the minimum transfer latency at 3.0 ms.

Our measurements show that overall, QPI provides a very low minimum transfer latency and therefore a lower latency for small transfers. However, PCIe 3.0 edges out QPI for larger transfers. For simultaneous reads and writes, PCIe 3.0 far outperforms QPI in bandwidth because PCIe has dedicated upstream and downstream channels, totaling an aggregate 13.6 GB/s. QPI has a single bidirectional bus, which we measured at 6.6 GB/s for sustained reads/writes.

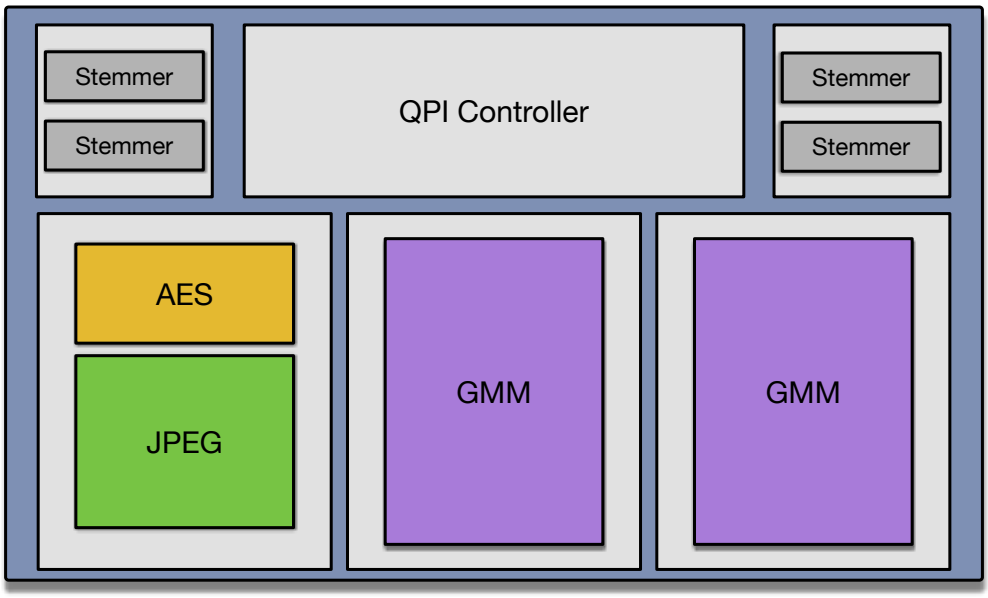


Figure 5.2: An example of FPGA fabric partitioning with multiple accelerators. Each GMM uses one tile, and AES and JPEG use hierarchical partitioning to use another tile.

5.2.3 FPGA Fabric Sharing

FPGA fabric can be shared both spatially (i.e., by programming multiple applications onto the same fabric) and temporally (i.e., by reprogramming the fabric over time to server different applications). While spatial sharing is fairly easy to do with the standard software tool chain, temporally sharing can be much more complex and expensive.

A naive way of sharing the FPGA fabric is to reprogram the entire fabric to reallocate resources for different applications. However, this approach results in significant reprogramming latency because the entire fabric needs to be reprogrammed, and the entire FPGA needs to be taken offline during reprogramming. To address this issue, partial reprogramming has been proposed by prior work. With the capability of partial reprogramming, recent works [29, 76, 95] use tile-based reprogramming to dynamically swap out bit streams at runtime. This significantly reduces the reprogramming latency because only a small region of the fabric needs to be reprogrammed, and also

keeps the other already programmed accelerators on the fabric available during the reprogramming.

Tile-based partial reconfiguration works well when the applications that are sharing the same FPGA fabric have similar amounts of utilization in terms of hardware resources (e.g., look up table, block memory). However, it can lead to inefficient hardware usage when the different modules have drastically different amounts of resource utilization, because the tile size has to accommodate the largest module. To address this issue, we apply the same tile-based partial reconfiguration technique in a *hierarchical* fashion, where partially reprogrammable regions are embebbed inside each other. In particular, we tile the fabric by first partitioning it into static regions and top-level reprogrammable tiles, where the static region contains the off-chip interface and static routing, and the top-level reprogrammable tiles are equally sized to accommodate the largest module. Within each tile, we apply the same tile-based reconfiguration technique recursively to continue partitioning resources, and break the recursion after two levels to avoid generating excessively high number of bit streams to store. For instance, suppose we have four different applications GMM, JPEG, AES and Stem, and the amount of resources need by each application is ranked as GMM > JPEG > AES > Stem. Figure 5.2 shows an example of how the fabric might be partitioned.

5.2.4 Workload Characterization

To understand how real-world datacenter applications perform on these CPU-FPGA systems, we implement four popular datacenter workloads on both systems and character their behaviors in terms of resource sharing.

5.2.4.1 Workloads

GMM – Gaussian Mixture Model is commonly used for automatic speech recognition in Intelligent Personal Assistant (IPA) services [59] (e.g., Apple Siri, Google Now, Amazon Echo). GMM is used to transcribe the words from a recorded sound waveform by comparing the speech against pre-trained models and returning the most likely result.

AES – Advanced Encryption Standard (AES) is a symmetric key algorithm used by SSL and TLS in almost all web services [21] (e.g., HTTPS). We use AES-128 with a 128-bit key length incorporating ten rounds with a ten stage pipeline, one stage per each round.

JPEG – JPEG is one of the most commonly deployed image compression algorithm used in image hosting services [62] (e.g., Facebook, Pinterest, Instagram). It compresses grids of 8x8 pixels by applying a discrete cosine transform to convert a 2D image to the frequency domain.

Stem – Word Stemming is used for document filtering in latency-sensitive web search [116] and IPA applications [59]. Stem finds the root word of an input word via a set of language rules, and it composes a significant portion of computation time simply because it must be executed for every word in a document.

5.2.4.2 Interconnect Sharing

Table 5.2: Workload Characteristics

Job	Function	Job Size	Data Size
AES	Encryption	128 KB document	128.00 KB
GMM	Speech processing	1 speech frame	0.11 KB
JPEG	Image encoding	256x256 image	192.00 KB
Stem	Word stemming	300 English words	4.69 KB

We characterize how the interconnect is being shared on real systems by profiling these applications on a CPU-only system, a QPI interconnected CPU-FPGA system,

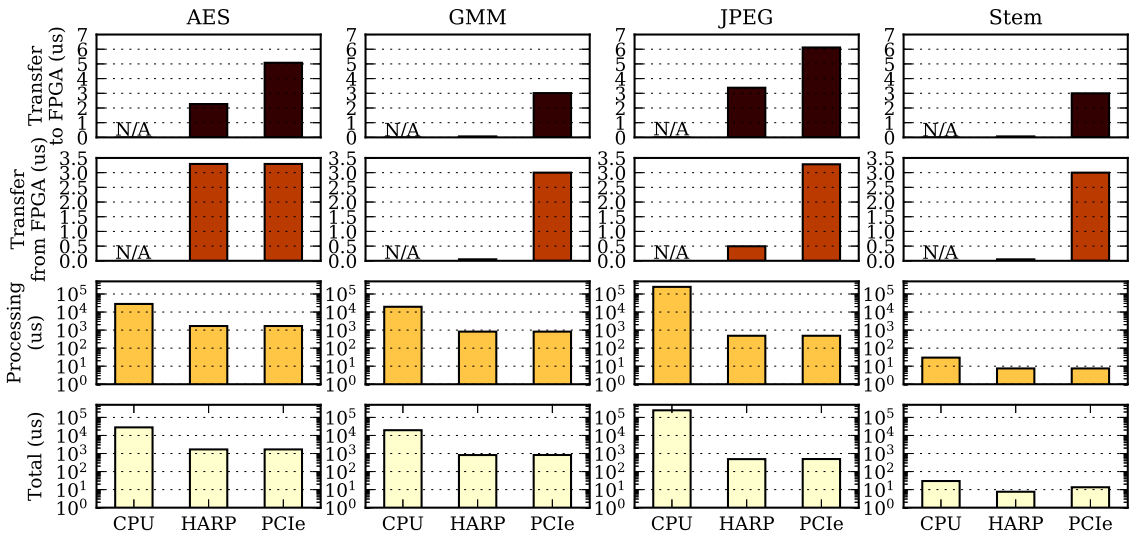


Figure 5.3: Average execution latency and data transfer latency for each workload. Averages were derived from 100k samples on each platform.

and a PCIe interconnected CPU-FPGA system. We run each application 100K times, and the results are shown in Figure 5.3. On both the QPI interconnected system (denoted as "HARP") and the QPI interconnected system (denoted as "PCIe"), the latency is dominated by the processing, while the communication latency from and to FPGA is negligible except for Stem on PCIe. Despite the communication overhead, all four applications get significant speedup (i.e., at least 2× and 27.7× on average) running on the FPGA. The data to compute ratio for Stem is much greater than the other workloads, and here we observe the impact of QPI's lower latency: the end-to-end latency for Stem is 1.8× faster on QPI versus PCIe.

Then we also characterize the amount of data each application needs to transfer to verify if sharing FPGA resource can potentially result in bandwidth saturation on the interconnect. Table 5.2 summarizes the amount of data needed by each application. Comparing to the bandwidth we characterize in Section 5.2.2, the interconnect will not be saturated even if the FPGA is processing tens of thousands of executions of these applications.

Given the low communication latency and the high available bandwidth, it is very

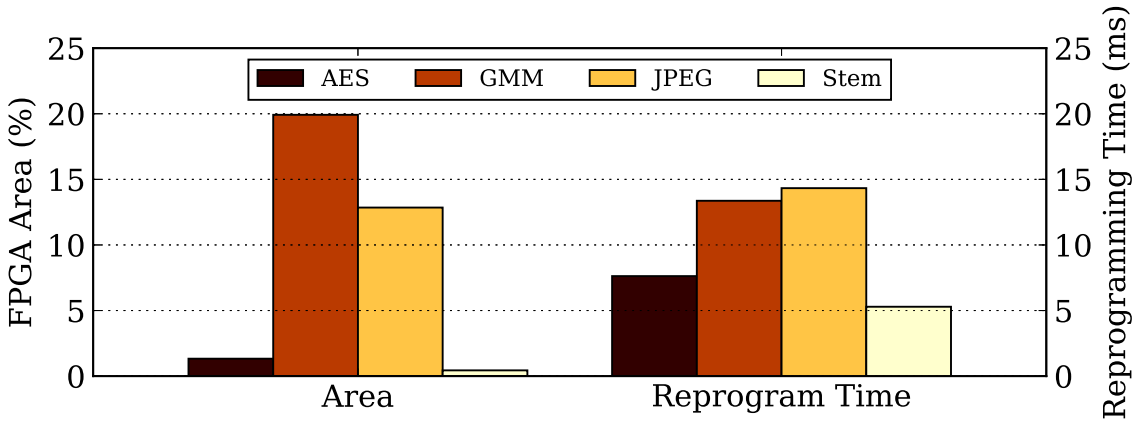


Figure 5.4: FPGA area (measured) and partial reprogramming time (derived from measurements of bitstream size and program time).

feasible to allow many applications running on the same FPGA without incurring significant performance overhead on the interconnects.

5.2.4.3 FPGA Fabric Sharing

Table 5.3: FPGA resource utilization for accelerators and interconnects

	Stratix V (Utilization %)		
Module	**LU**	**BM**	**DSP Blocks**
AES	1.33	0.65	0.00
GMM	10.2	0.03	19.9
JPEG	12.9	0.01	0.00
Stem	0.44	0.00	0.00
QPI Interface	28.7	1.97	0.00
PCIe Interface	6.91	1.98	0.00

As we mentioned in previous sections, FPGA fabric can be shared in two ways: spatial sharing and temporal sharing.

Spatial sharing: To characterize spatial sharing, we synthesize the FPGA implementation of each workload for both the QPI and PCIe systems and tabulate the FPGA utilization (percent of FPGA resources used by the workload) in Table 5.3, in which LU denotes look up table, BM denotes block memory, and DSP denotes digital signal processing blocks. None of the applications consumes more than 20%

of one single type of resource, which means several applications can be programmed on the same FPGA fabric by sharing the resources spatially. In addition, we observe that applications have drastically different characteristics in their hardware resource utilization, which means sharing the FPGA fabric among various applications can improve the resource utilization (e.g., applications with high LU utilization can share the FPGA fabric with applications with high DSP block utilization to achieve high utilization on both LU and DSP blocks).

Temporal Sharing: The FPGAs can be reprogrammed from on-board flash memory or JTAG. Flash memory is very slow to write data into, but the flash memory can hold many configuration bit streams and reprogram the FPGA from pre-stored bit streams much faster than JTAG. To study temporal sharing, we partial reprogram the FPGA fabric using bitstreams stored in the on-board flash memory and measure the reprogramming time for all four applications. Figure 5.4 presents the reprogramming time, which is on the order of 10s of milliseconds. Although this is still relatively high comparing to the latency of running applications on FPGA as shown in Figure 5.3, it is similar or sometimes lower than the latency of running applications on CPU. Therefore, dynamically reprogramming the FPGA to enable temporal sharing of the FPGA fabric can potentially improve the application performance.

In summary, spatial sharing FPGA fabric among applications can further improve FPGA resource utilization due to the diversity in application characteristics, and temporal sharing by reprogramming the fabric dynamically is fast enough to potentially improve application performance.

5.3 System Architecture

Prior work statically provision FPGA resources for single workloads [121], which does not cope with the dynamic nature of modern datacenter workloads and thereby heavily underutilizes the FPGA resources as shown in Figure 1.3. To improve resource

utilization, co-location of applications on the same servers has already been commonly employed to enable resource sharing in modern datacenters [143, 161, 52]. Several key characteristics we discovered through real system characterization in Section 5.2 of the emerging tightly coupled CPU-FPGA systems present us an unique opportunity to leverage FPGA as a shared resource among applications to improve FPGA resource utilization.

- **Low Communication Overhead:** Both QPI and PCIe provide low latency and high bandwidth communication between CPU and FPGA, which allows different applications to share the same interconnect without incurring significant communication latency.
- **Fine-Grained Spatial Resource Sharing:** Different workloads have drastically different characteristics in FPGA resource utilization as shown in Table 5.3, so we can achieve high resource utilization by sharing FPGA resource spatially at fine granularity among workloads (i.e., high LU utilization applications can share the same FPGA with applications with high DSP block usage).
- **Rapid Partial Reprogramming:** With the capability of rapidly partial reprogramming the FPGA fabric, we can dynamically reallocate resources among the applications that are sharing the same FPGA to cope the dynamism of modern datacenter workloads.

Therefore, we present a novel datacenter architecture, in which FPGAs are shared among different co-locating applications in both spatial and temporal fashions.

5.3.1 Cluster-Level Architecture

To improve resource utilization, it is a common practice for large datacenter operators like Microsoft [161], Google [143], and Facebook [52] to employ co-locations of different applications (e.g., interactive services, and batch processing workloads).

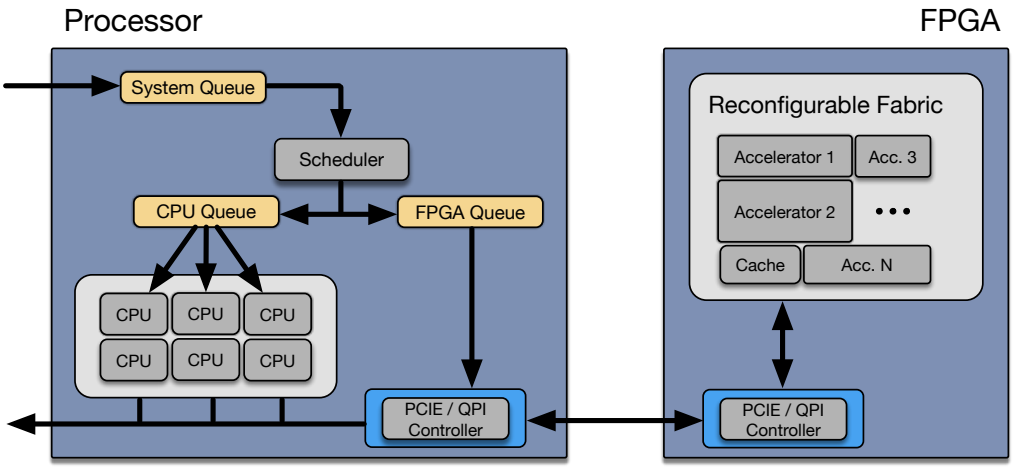


Figure 5.5: Server-level query flight diagram for Slingshot. The Slingshot scheduler estimates service time (including queuing, transfer and execution time) for the FPGA and CPU, then schedules the query.

Similarly, allowing different applications to share the same FPGAs can significantly improve resource utilization as illustrated in Figure 1.3. Therefore, we allocate a pool of FPGA-CPU systems as *acceleration servers* in each cluster to serve as a shared resource, which can be utilized by the *application servers* that serve specific applications.

When a user query/job hits our data center, the application servers will first house the top-level application (e.g., web search, intelligent personal assistant (IPA), and MapReduce batch processing jobs). During the execution of the query/job, there are certain components (e.g., computational expensive kernels like GMM and Stem in IPA applications [59]) can be accelerated by FPGAs, which we refer as a *microservice query*. These microservice queries will be sent over to the shared acceleration servers for processing. Every acceleration servers in the shared pool are logically identical, which means each server is capable of executing any microservice query.

5.3.2 Server-Level Architecture

Each server in the shared acceleration pool runs a CPU-FPGA system, where both the CPU and the FPGA accelerator are shared among the microservice queries issued by various applications.

Figure 5.5 presents a flow diagram of how each microservice query is being handled at server-level. When a microservice query is issued by application servers, the query is put into a system queue on the acceleration server. Then a scheduler will schedule the query to CPU or FPGA by moving the microservice query to the corresponding queue, which introduces the first question we need to answer.

- **When should a query be scheduled to FPGA?** Although FPGA execution can often provide speedup over CPU execution for a single query as shown in Section 5.2, making this scheduling decision dynamically in a complex queueing system is quite challenging. For example, scheduling too many queries to FPGA can result in long queueing latency, or even saturate the interconnect bandwidth, in which case the CPU becomes the preferential target for scheduling. In addition, the FPGA may not already have the scheduled microservice programmed on the fabric, causing additional reprogramming overhead.

When a microservice query is scheduled to the FPGA accelerator, the query will be pushed into the FIFO queue, which waits on both FPGA interconnect and accelerator to become available. If the microservice has not been programmed on the FPGA yet, a repartitioning manager needs to reprogram the FPGA fabric, which raises the second question we need to answer.

- **When should the FPGA fabric be repartitioned?** Reprogramming FPGA fabric to handle certain types of microservice queries incurs overhead in short term, but may improve overall system performance in longer term. For exam-

ple, increasing the resources allocated for GMM may improve overall system performance when the load of IPA service is increasing.

Note that these two questions are not independent from each other. In fact, they actually have complex interactions. Reprogramming decisions can affect scheduling decisions due to the resource availability (e.g., more queries can be scheduled to the FPGA when increasing amount of resources is allocated for the corresponding microservice). On the other hand, scheduling decisions can also affect reprogramming decisions (e.g., repartitioning more resources for a microservice can improve system performance when there are many queries scheduled to FPGA). Care must be taken to coordinate the two components.

5.4 Resource Management

To answer the two questions raised in Section 5.3.2, we introduce a resource management system, Slingshot, that complements the dynamism of modern datacenter workloads and better utilizes datacenter resources. Specifically, we present a query scheduling algorithm in Section 5.4.1 to decide *when a query should be scheduled to FPGA for acceleration*, and a repartition scheduling algorithm in Section 5.4.2 to choose *when and how to repartition the FPGA fabric*.

5.4.1 Query Scheduling

The main goal of query scheduling is to identify the resource that will offer the lowest service time. The key idea of Slingshot's query scheduler is to *use latency estimation to estimate the lowest-latency resource*. The following sections discuss the challenges and implementation of our scheduler.

5.4.1.1 Challenges

The Slingshot query scheduler has the option of assigning a query to any available computation resource (CPU cores or FPGA accelerator units). Based on the characterization from Section 5.2, we find that the FPGA is usually the optimal choice, if available. The FPGA accelerators have a substantially smaller service time, which significantly increases the machine's maximum throughput provided there are accelerators available for the given query type. However, the FPGA still faces queuing latency issues under heavy load. If too few acclerators are available or the queue is too long, queuing effects can raise FPGA service time to exceed the CPU service time.

Previous works have explored estimating latency as an algorithm for resource scheduling. Craeynest et al. and Duggan et al. leverage profiling information to estimate latencies [138, 38], however profiling is impractical with accelerator units that constantly change and have on the order of $O(mn^2)$ possible accelerator combinations (where n is the number of accelerator types and m is the number of accelerators that can fit on the FPGA simultaneously). The analytical models of Wu et al. and Chandramouli et al. [152, 23] suffer similar problems handling large numbers of possible reconfiguration states. Because the available resources in a dynamically allocated FPGA change over time, our platform is fundamentally different than prior works and requires new techniques to maximize resource utilization.

5.4.1.2 Design

When performing query scheduling, Slingshot estimates the end-to-end latency based on standalone job profiling (Section 5.2) to predict the latency of issuing a query to a target compute resource. The latency model is then broken down into into three components: processing latency, transfer latency, and queuing latency (Table 5.4).

One of the largest benefits of FPGA accelerators is that the processing time can be predicted to an exact number of cycles. Data transfer to the FPGA data transfer is also highly predictable using both PCIe and QPI, and does not change as the FPGA is reconfigured. Thus, the total FPGA latency is highly predictable for a single query's end-to-end latency. The queuing latency is much more variable because our system pipelines data transfer and query processing. In the worst case, the queuing time will be no greater than the sum of all of the latencies of queries in the queue. However, the real queuing time is much smaller than the worst case because of pipelining and parallelization, but it is also highly variable because individual queries can contend processing or interconnect resources from other queries, making the queuing latency highly dependent on the workload composition and ordering of the queue.

Table 5.4: Latency estimation model components (Bidirectional arrows represent round-trip).

Latency Term	**Symbol**	**Representation**
Process (CPU)	t_{proc}	memory $\rightleftharpoons$ processing
Process (FPGA)	t_{proc}	FPGA cache $\rightleftharpoons$ processing
Transfer	t_{trans}	memory $\rightleftharpoons$ FPGA cache
Queue	t_{queue}	queue insert $\rightarrow$ queue pop

For a first-order latency estimation given n workloads, the latency of a given query t_{query} can be estimated as:

$$t_{query} = t_{proc,query} + t_{trans,query} + t_{queue}$$

$$t_{queue} \approx \sum_{i=1}^{n} (t_{proc,i} + t_{trans,i}) \times m_i$$

where m_i is the number of queued queries for workload i in a specific queue. Our query scheduler can keep track of m_i for the CPU queue and the FPGA queue by incrementing a counter when the queries enter the queue and decrementing the counter when the queries exit the queue, which leads to an extremely small constant overhead per query.

However, this model assumes each query in the queue is processed serially. Queries executed on CPU can be distributed to each available core, reducing the queue time significantly over serial execution. On the FPGA side, data transfers can be pipelined with processing while processing is parallelized across multiple accelerator units. As a result, this naive model performs poorly in a complex system. Therefore, Slingshot requires a second order model to account for the specific architecture of a system with multiple execution paths.

Slingshot's second order model focuses on the average case for queries in the queue. Because the difference between the service paths on CPU and FPGA is significant, Slingshot must consider different models for each. For the CPU, each query can be serviced by any processing resource. Therefore, the latency can be estimated by multiplying the average query latency by the number of processing batches it will take to process the entire queue:

$$t_{queue,CPU} \approx \left\lceil \frac{m}{r} \right\rceil \frac{1}{m} \sum_{i=1}^{n} t_{proc,i} \times m_i$$

where r is the number of processors, and m is the total number of queries in the CPU queue. On the FPGA, there are more factors to consider. Each query can only be serviced by an accelerator of the approperiate workload type, unlike the CPU. Because different workload types can be parallelized across each type's accelerator units, however, the queue processing latency will only be limited by the total processing latency of the longest latency workload. In addition, the data transfer can be pipelined with the data processing, so the queue time can be estimated by whichever is longer, the total transfer time or total processing time:

$$t_{\text{queue,FPGA}} \approx \max \left(\sum_{i=1}^{n} t_{trans,i} \times m_i, \max_i \left(\left\lceil \frac{m_i}{r_i} \right\rceil t_{proc,i} \right) \right)$$

where r_i is the number of processing resources for workload i. These approximations

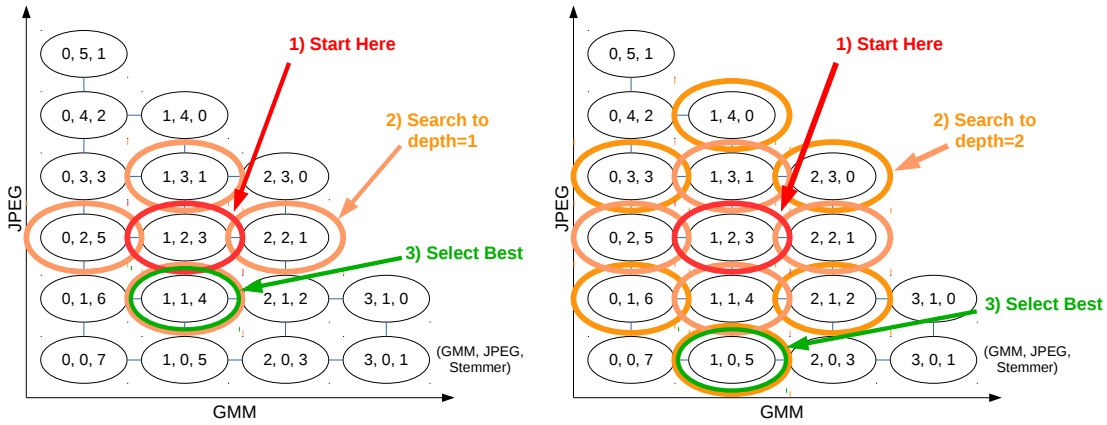


Figure 5.6: Example configuration space of an FPGA with 3 accelerator options, searching to depth=1 (left) and depth=2 (right). Each node represents a possible configuration of accelerators that would fit on the FPGA.

assume the overall queue for the CPU will only be bottleneck on processing resources, whereas the FPGA can bottleneck on either interconnect bandwidth or processing resources.

In a real system, the actual bottlenecks will be a combination of resources and bandwidth at different points during execution. However, the only way to effectively estimate the latency to a higher degree would require iterating through the entire queue and estimating the bottleneck on a query-by-query basis. While this model would likely offer increased accuracy, we show in Section 5.5 that the second order latency model offers a high degree of accuracy with a low, queue-length-independent overhead.

5.4.2 Repartition Scheduling

The main goal of repartition scheduling is to properly match the allocated FPGA accelerators to the current datacenter demand. The key idea of Slingshot's repartition scheduler is to *use the recent history of queries to estimate demand and then evaluate the benefits of repartitioning.* The following sections discuss the challenges and implementation of our scheduler.

5.4.2.1 Challenges

The Slingshot resource manager is unique to FPGAs due to two key factors: (1) the fine-grained control over the hardware resources and (2) the high and variable resource scheduling delays caused by reprogramming accelerator units.

FPGAs have an inherently *high-risk, high-reward tradeoff* when repartitioning resources. Accelerator units offer very fast processing time compared to CPUs even with the memory transfer overhead; however, the time to switch from one workload type to another is two orders of magnitude higher on an FPGA than it is on a CPU [39, 85]. The reprogramming time (milliseconds) is also much higher than the compute latency (microseconds) so Slingshot's resource manager cannot make decisions based upon individual queries, and must weigh the resource cost of switching against the benefits for switching for a distribution of queries.

5.4.2.2 Design

The first step in making a decision to reconfigure the FPGA fabric is to understand the domain of configuration states. Because of the restrictions applied to accelerator placement in Section 5.2.3, there is a finite number of configuration states to which the FPGA can be reprogrammed. After eliminating impossible and underutilized states, we can create a domain of possible configuration states that lie an a Pareto-optimal curve of peak throughput for each workload. This curve lies in an n-dimensional space, where n is the number of workloads.

The next step in the algorithm is to perform a search for the Pareto-optimal configuration node based on the current workload demand. Because searching the entire configuration domain becomes unwieldy as the number of nodes increases, we perform a depth-limited breadth-first search where we originate at the current configuration node and traverse k edges. Figure 5.6 displays an example of a search over a domain of 3 workloads with a search depths of 1 and 2, however the algorithm

can be performed with an arbitrary depth. At each node, the scheduler estimates the benefits and costs of changing the configuration state to the new node. The benefits are calculated by subtracting the current node's normalized throughput from the target node's estimated throughput, and the costs are estimated by multiplying the drop in throughput by the reconfiguration time. If the benefits exceed the costs, the repartitioning schedules will attempt to initiate a partial reconfiguration. In the event that queries are queued up to run on the FPGA but the reconfiguration process would remove all applicable accelerators, the repartitioning scheduler will wait until the queue is clear and also notify the query scheduler not to schedule more queries of that type on the FPGA.

5.5 Evaluation

To evaluate our system, we first present the details of our evaluation methodology in Section 5.5.1. Then we quantify the prediction accuracy of our latency estimation used for managing resources in Section 5.5.2. Last, we evaluate the full system performance in terms of both QoS improvement and throughput improvement in Section 5.5.3.

5.5.1 Evaluation Methodology

5.5.1.1 Experimental Setup

We evaluate Slingshot using BigHouse [105], an event-driven, queuing-based datacenter simulator that uses workload characteristics to synthesize an event trace. We use BigHouse for two key reasons: firstly, a BigHouse allows us to vary system parameters such as FPGA area and bandwidth and project how Slingshot can apply to future systems or systems with different constraints. Secondly, current FPGAs are designed to hold only a small number of bitstreams in the on-board flash memory.

We do not need significantly more memory: our test system requires around 4 copies of each partial-board accelerator bitstream for the Stratix V (approximately 200 MB of memory in total), however the highest capacity Stratix V dev board currently only comes equipped with 64 MB of flash. For these reasons, we modify BigHouse to suit our needs. We incorporate several new features, based on measurements from real hardware, to allow for precise modeling and analysis of Slingshot running on a CPU-FPGA platform such as adding interconnect and accelerator devices and queues.

Modeling system architecture - Our model is built from the architecture discussed in Section 5.3.2. We model the incoming query queue (system queue), as well as the CPU, FPGA, and interconnect queues as described in Section 5.3. When a query arrives, it is placed in the main system queue until the query scheduler can assign it to either the CPU or the FPGA. If the query is assigned to the CPU, it is dispatched to the CPU queue to wait for an available CPU core. Otherwise, it is dispatched to the FPGA queue until both the interconnect is available and the target accelerator is free.

Modeling CPU latency - We model CPU execution time from creating a cumulative distribution function (CDF) of 100k service times measured on real hardware. This CDF captures variability from compute, memory, cache, and other variation sources.

Modeling FPGA latency - We model the FPGA latency through the combination of an interconnect model and a processing latency model. The processing latency model is a CDF composed of 100k measurements and captures the (very low) processing variability. The interconnect model is composed of the 1000 interconnect transactions measured on our real system, and captures the variability associated with memory access at the host machine side and data transfer latency from the host machine to the FPGA over the interconnect.

Modeling workload traffic pattern - To accurately model the inter-arrival

rate of queries, we use inter-arrival time distributions from a production Google web cluster [103]. Our load patterns (discussed in the next section) include two synthetic workloads and a real web datacenter query trace from search provider SoGou [147].

To simulate a wide range of load patterns hitting a datacenter, we extend BigHouse with the ability to update the the query-per-second (QPS) parameter of the query-arrival stochastic process on the fly. We also alter BigHouse to enable transient simulation that run for a specific amount of time, in addition to steady state load patterns that converge as dynamic reconfiguration of resources in the face of varying traffic cannot produce a steady state service distribution.

Modeling scheduling and repartitioning overhead - To model the dynamic aspects of our system, we implement both the query scheduling and repartitioning algorithms from Section 5.4. For both query scheduling the FPGA repartitioning, we include the overhead of making and executing the decisions. For query scheduling, we model the latency overhead incurred by the query scheduler by adding the average execution time of the scheduler measured in real system tests.

To account for FPGA repartitioning, we model the reconfiguration overhead by using the reprogramming time data shown in Figure 5.4. Our simulator mimics the real-system reconfiguration process: we take the reconfiguring accelerators offline for the duration of the corresponding reprogramming time, and then bring the reprogrammed accelerator online once reprogramming is complete. During this period of time, the part of FPGA resource involved in the reprogramming is not available to process queries. We measured the average repartition scheduling latency at 72 μs, with the 99-th percentile of 76 μs.

5.5.1.2 Workloads

To evaluate our systems, we use 3 different load patterns that showcase different behaviors representative of a real datacenter environment (Figure 5.7). The first load

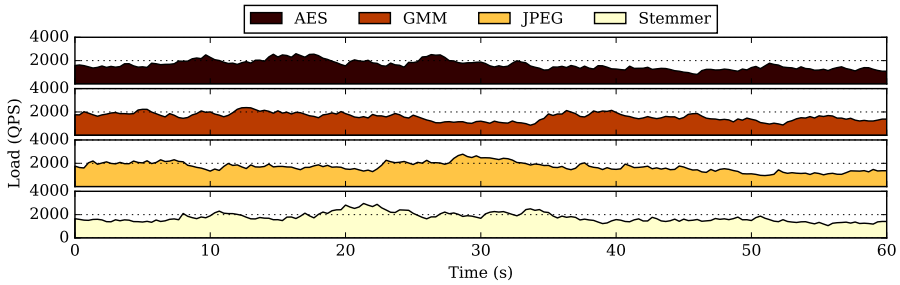
is a trace over 60 seconds of a relatively "slow changing" load. The load is composed of approximately 400k queries with the instantaneous load ranging from 5k to 8k Queries Per Second (QPS). This pattern was selected because it represents the low end of dynamic load changes in a datacenter, such as off-peak times in the SoGou dataset.

The second load is an evaluation over 60 seconds of a relatively "fast changing" load. The load is composed of approximately 390k queries with the instantaneous load ranging from 5k to 8k QPS. This pattern was selected because it represents the high end of dynamic load changes in a datacenter, such as peak times in the SoGou trace.

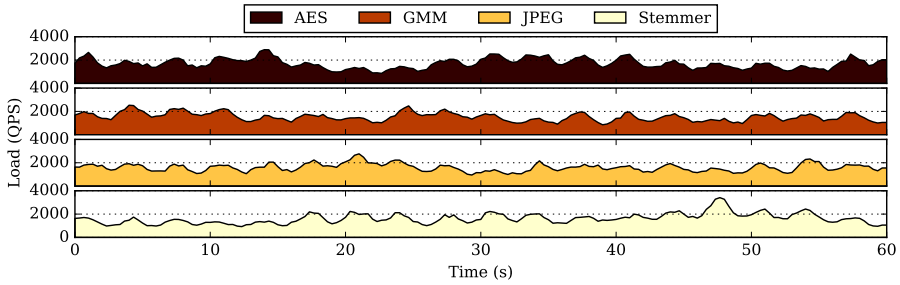
The third load is a based off of a real datacenter request trace obtained from the BigDataBench dataset [147], which was sourced from web search provider SoGou. The load pattern is composed of approximately 400M queries over 24 hours. The real 24-hour trace was converted into a load pattern by taking the peak instantaneous load in the trace and scaling it to 8k QPS after ignoring outliers to maintain consistency with the other load patterns (there are in total 10 data points that lie above 8k QPS after scaling, with each datapoint lasting 1 second). This load pattern displays the traditional datacenter diurnal load cycle, as well as the real noise associated with datacenter traffic. We have performed curve smoothing on the trace shown in Figure 5.7 in order to observe the load pattern, however we performed no alterations to the data used to run our experiments.

5.5.2 Scheduling Prediction Accuracy

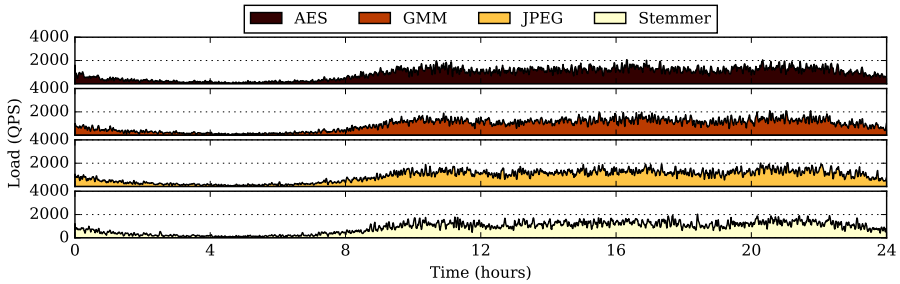
To evaluate our latency prediction model, we run our fast and slow load patterns on our QPI and PCIe systems for both dynamic (partially-reconfiguring) and static (non-reconfiguring) systems. We evaluate the prediction accuracy by sampling every 100 queries and comparing the execution latency on the two devices. Results show that



(a) Slowly changing load pattern.



(b) Quickly changing load pattern.



(c) Load trace from SoGou [147]

Figure 5.7: Load patterns used in evaluation.

our proposed query scheduler is able to correctly predict the lower-latency resource for more than 96% of queries. In the cases where our algorithm does not make the correct prediction, the difference in actual latencies is less than 7%. Our query scheduler is extremely effective at correctly predicting latencies at lower loads due to the length of the queue being relatively small. At higher loads, the queuing time becomes harder to predict without precise information about the composition and ordering of the queues because queries will contend for resources.

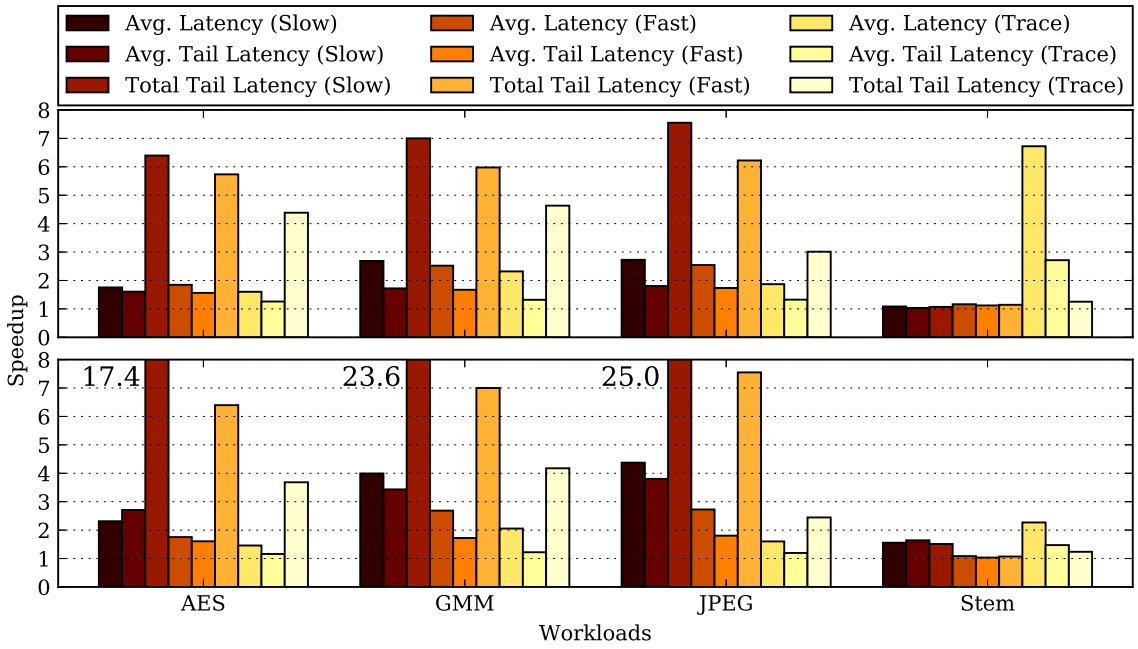


Figure 5.8: QoS improvement for dynamic scheduling over static scheduling on a simulated FPGA system with QPI (top) and PCIe (bottom)

5.5.3 Full Systems

Using our system, we evaluate 2 key metrics across both our HARP and PCIe platforms: 1) We evaluate the QoS improvements over a static system by implementing dynamic reprogramming for each load pattern. 2) We evaluate the per-machine efficiency improvement by measuring the additional load capacity of a dynamically reprogrammed system over a statically programmed system. For each of these experiments, both our baseline systems and our evaluation systems use the specifications from Section 5.2. Our evaluation compares the systems using a statically programmed design (similar to prior work) to the systems using our dynamic reprogramming algorithms from Section 5.4. Our static systems are partitioned such that the FPGA resources are maximized and each workload's peak throughput is matched as closely as possible (i.e. the peak GMM, JPEG, AES, and Stem query throughputs match).

5.5.3.1 QoS Improvement

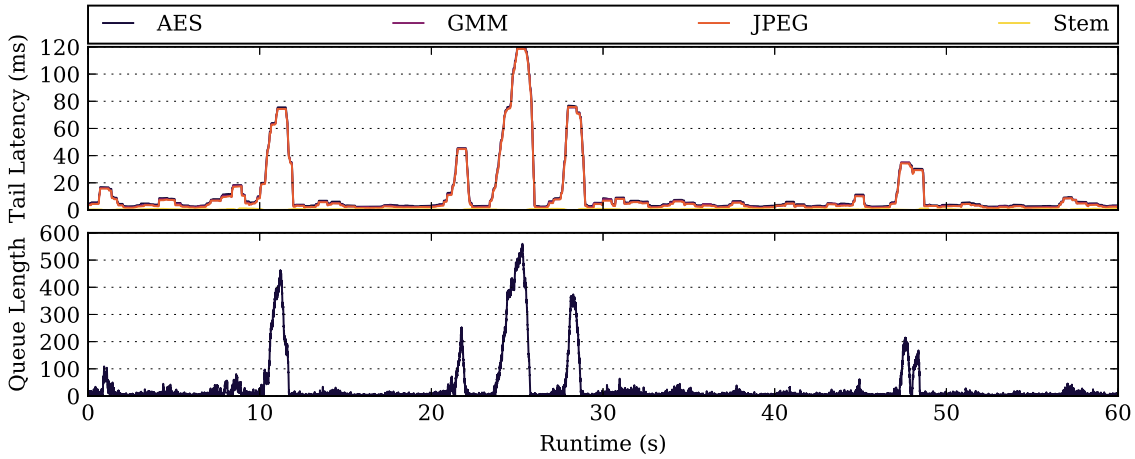
In Figure 5.8, we show the improvement of our dynamic system over a baseline static system for QPI and PCIe. To measure the QoS of these systems, we employ 3 metrics to evaluate latency. First, we use average latency, which is the arithmetic mean of the latency across the entire experiment. We use total tail latency, which is the 99% latency of the entire experiment. Lastly, we use a metric called average tail latency. Tail latency is usually used in a static context, where the tail latency is a function of a static load (e.g. fixed queries per second). Because the experiment employs dynamic loads and systems, we also utilize an average tail latency metric, where the 99% latency is sampled over a 5 second window and stepped across the experiment at 5 ms intervals. We then take the arithmetic mean of these tail latencies to give a better view of the short term worst case latency as the load changes.

Figure 5.8 shows that the dynamically programmed system improves QoS across all of our metrics. Average latency is improved by 1.9× on our QPI system and 2.3× on our PCIe system (geometric mean). Average tail latency is improved by 1.5× for QPI and 2.0× for PCIe, and total tail latency is improved by 4.2× for QPI and 7.0× for PCIe.

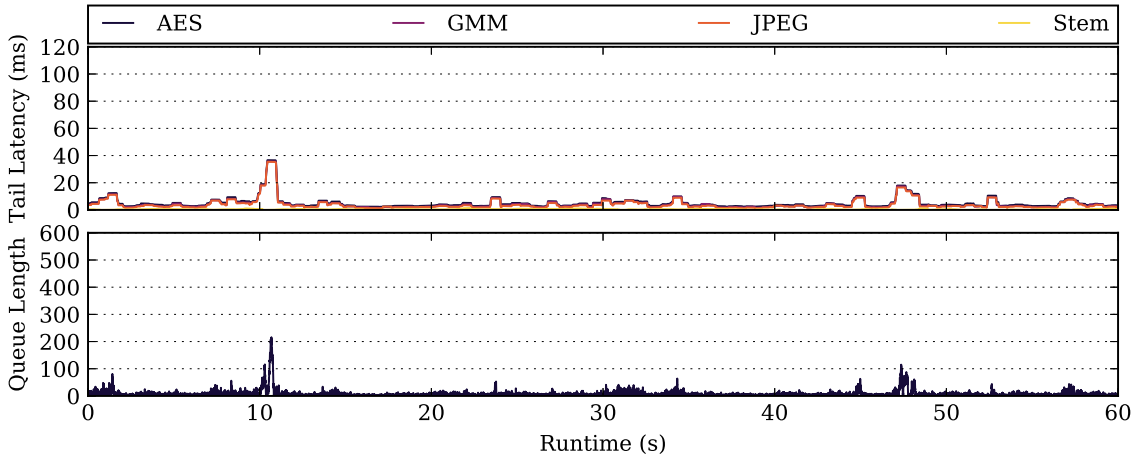
From these results, we observe that QoS improves across the board, but the orders of magnitude processing speedup seen in Section 5.2 do not translate proportionally into QoS improvement. We find that the major contributor to QoS degradation stems from transfer queue blockages: when a relatively large number of queries arrive in a short interval, the FPGA interconnect must transfer the queries one at a time. Our dynamic system is able to relieve a majority of slowdowns due to processing resource allocation, which are seen on the static system.

An example of this scenario is shown in Figure 5.9, which shows the windowed tail latency and total queue lengths for the fast changing workload with and without Slingshot. At about 11 seconds into the workload, both systems encounter a latency

spike, which we found was caused by a sharp increase in the GMM queries per second. Slingshot (Figure 5.9b) was able to reallocate a group of Stem and AES modules for an additional GMM accelerator, whereas the static system could not. The queue length for the dynamic system peaks at 214 queries and then declines as the additional accelerator is able to process new queries. The static system is not able to allocate new units, so although it begins distributing queries to the CPU, the tail latency spikes as the load increases.



(a) Windowed tail latency and total queue length without Slingshot.



(b) Windowed tail latency and total queue length with Slingshot.

Figure 5.9: Comparison of the fast changing workload on HARP without Slingshot (a) and with Slingshot (b).

5.5.3.2 Load Capacity Improvement

For our second experiment, we examine the improvement in per-machine throughput gained from Slingshot when compared to a static system. We linearly scale the load of our slow and fast load patterns, and then compare the total tail latency of the scaled load to a static system with an unscaled load. Figure 5.10 shows a sweep of our platforms and load patterns normalized to a static system baseline. The red line represents parity in total tail latency to the static baseline. From the figure, we can achieve up to a 26% increase in load for our HARP system compared to the static baseline, and we can achieve up to a 70% increase in load for our PCIe system while maintaining the same QoS as our static baseline.

Through these experiments, we observe that Slingshot can effectively use resources to accommodate dynamic datacenter load. We observe that PCIe both obtains better speedup over the static baseline than HARP and also can support more load than HARP for a given QoS, despite HARP having an extremely low latency interconnect. As shown in Figure 5.3, the difference in transfer latency is negligible compared to the total processing time. The major contributing factor to the performance increases are due to the differences in area: the PCIe system has a hard PCIe controller block that consumes only 6.91% of the FPGA fabric, whereas HARP uses approximately 29% of the fabric for the communication interface. This difference in area allows us to fit an additional top-level tile on the FPGA, which can fit another GMM or JPEG accelerator, as well as several additional AES or Stem accelerators. These additional accelerators allow Slingshot more flexibility in partitioning its resources and to better accommodate its load, especially during peak loads.

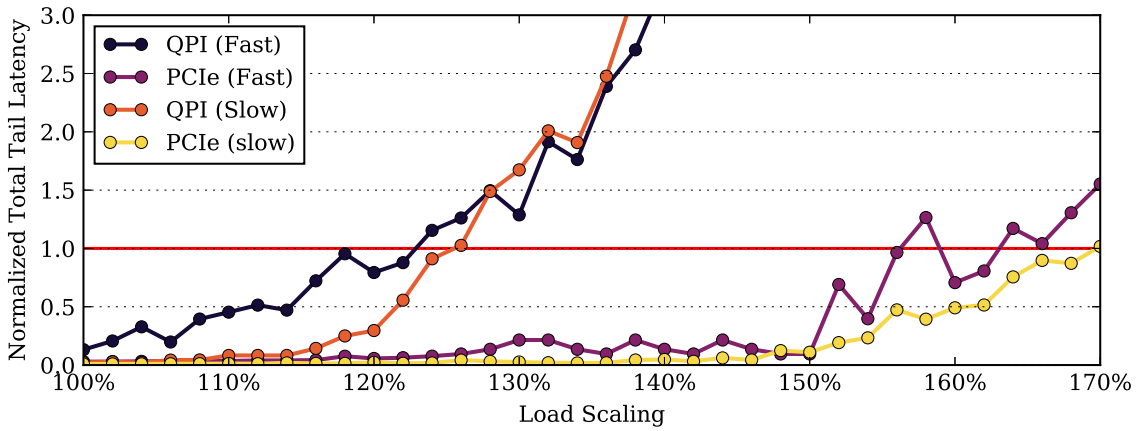


Figure 5.10: Load improvement of Slingshot-equipped servers over their static baselines on the same platform. A normalized total tail latency of 1 indicates QoS parity with the baseline.

5.6 Summary

In this chapter, we presented a characterization of two real FPGA accelerated servers: a PCIe attached FPGA and Intel's HARP prototype with an FPGA attached to a server-class Xeon CPU over QPI, a low-latency memory-coherent bus. We characterize the performance and resource utilization of four real datacenter workloads accelerated on these real FPGA systems, and use our characterization to present Slingshot, an FPGA-accelerated datacenter design. We model Slingshot in the BigHouse simulator suite and explore real and synthetic load datacenter patterns. We find that dynamic scheduling offers significantly increased average throughput and increases average QoS metrics by 1.7× to 8× over a static FPGA management system, or provides a 23-70% increase in server load capacity while maintaining the same 99th percentile tail latency as a static management system.

CHAPTER VI

Conclusion

When building and running modern, massive-scale datacenters, power and energy have become first-class concerns. They not only impact the cost of building the infrastructure and the electricity bill, but also directly relates to the computation capacity a datacenter can deliver, which plays a critical role in gaining profit and ensuring the quality of services.

This dissertation investigates the power and energy utilization of datacenters running modern cloud and machine learning workloads, and identifies the sources of inefficiency at three levels of the datacenters: datacenter level, server level, and computing fabric level. It then provides insights and approaches to mitigate the inefficiency at each of these levels.

At the datacenter level, I identify the power budget fragmentation problem found in three Facebook production datacenters. To solve the fragmentation problem and improve the power efficiency, I leverage the knowledge of the temporal heterogeneity of power consumptions of different workloads, and design the SmoothOperator framework to unleash the once wasted power budgets. Firstly, SmoothOperator applies workload-aware service instance placement technique to distribute service instances with similar power consumption pattern across the datacenter, unlocking the fragmented power budget. It then learns from the history of power consumption patterns

and applies server-conversion-based dynamic power profile reshaping runtime in the production environment to further improve the power budget utilization at datacenter level.

At the server level, I characterize the latency distributions two most commonly deployed workloads, and found that different types of queries not only have different latency distributions, but also are impacted differently by voltage and frequency scaling. My insight is that, we can achieve better tail latency without increasing the overall energy consumption when there exist tail queries – the types of queries whose latency not only are highly likely to fall into the tail of the overall distribution, but also can be significantly impacted by voltage and frequency of the core. I design the Adrenaline methodology that adjusts the voltage/frequency at query-level granularity to rein in the tail latency as well as to save energy. Adrenaline pinpoints these tail queries and proactively raise the V/f setting for them to reduce their latency, which reins in the tail of the entire distribution. At the same time, it leaves those "already fast enough" queries running at lower V/f setting, trading the mean latency for lower overall energy consumption.

Lastly, I find that the mainstream CPU-centric datacenters are overlooking significant energy saving opportunity. These general purpose computing units are designed for highest flexibility but not for achieving high energy efficiency. On the contrary, application specific accelerators have limited programmability in exchange for significantly higher efficiency. Among these accelerator platforms, FPGA strikes a balance among programmability, performance, and efficiency. To explore how FPGA adapts to the highly dynamic cloud environment, I investigate the partial reconfigurability and the hierarchical partial reconfigurability of modern FPGAs. I characterize the performance and resource utilization of four real datacenter workloads accelerated on these real FPGA systems, and use our characterization to present Slingshot, an FPGA-accelerated datacenter design. With all the real system measurements, I

model Slingshot in the BigHouse simulator suite and explore real and synthetic load datacenter patterns. We find that dynamic scheduling offers significantly increased average throughput and increases average QoS metrics, and improves server load capacity while maintaining the same 99th percentile tail latency as a static management system.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Earnings Buzz: Twitter Inc (TWTR), Facebook Inc (FB), LinkedIn Corp (LNKD). http://www.ibtimes.com/earnings-buzz-twitter-inc-twtr-facebook-inc-fb-linkedin-corp-lnkd-2028461.

[2] Facebook Newsroom. http://newsroom.fb.com/company-info/.

[3] Introducing data center fabric, the next-generation Facebook data center network. https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[4] Open Compute Project. http://www.opencompute.org.

[5] Y. Ago, A. Inoue, K. Nakano, and Y. Ito. The parallel fdfm processor core approach for neural networks. In *Networking and Computing (ICNC), 2011 Second International Conference on*, 2011.

[6] B. Aksanli, E. Pettis, and T. Rosing. Architecting Efficient Peak Power Shaving Using Batteries in Data Centers. 2013.

[7] S. R. Alam, P. K. Agarwal, M. C. Smith, J. S. Vetter, and D. Caliga. Using fpga devices to accelerate biomolecular simulations. *Computer*, 2007.

[8] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 2009.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

[10] Z. K. Baker and V. K. Prasanna. Efficient hardware data mining with the apriori algorithm on fpgas. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, 2005.

[11] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 2013.

[12] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*.

[13] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[14] A. Beloglazov, J. Abawajy, and R. Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 2012.

[15] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[16] A. A. Bhattacharya, D. Culler, A. Kansal, S. Govindan, and S. Sankar. The need for speed and stability in data center power capping. *Sustainable Computing: Informatics and Systems*, 3(3):183–193, 2013.

[17] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars. Enabling fair pricing on hpc systems with node sharing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013.

[18] R. Buyya, A. Beloglazov, and J. Abawajy. Energy-efficient management of data center resources for cloud computing: a vision, architectural elements, and open challenges. *arXiv preprint arXiv:1006.0308*, 2010.

[19] O. Callanan, A. Nisbet, E. Özer, J. Sexton, and D. Gregg. Fpga implementation of a lattice quantum chromodynamics algorithm using logarithmic arithmetic. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005.

[20] J. Cardoso and M. Hübner. *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*. 2011.

[21] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, L. Sitaram, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *Microarchitecture, 2016 IEEE International Symposium on (MICRO)*, 2016.

[22] S. Chalamalasetti, M. Margala, W. Vanderbauwhede, M. Wright, and P. Ranganathan. Evaluating fpga-acceleration for real-time unstructured search. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, 2012.

[23] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos. Accurate latency estimation in a distributed event processing system. In *2011 IEEE 27th International Conference on Data Engineering*, 2011.

[24] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[25] H. Chen, C. Hankendi, M. C. Caramanis, and A. K. Coskun. Dynamic server power capping for enabling data center participation in power markets. In *Proceedings of the International Conference On Computer Aided Design (ICCAD)*, 2013.

[26] Q. Chen, H. Yang, J. Mars, and L. Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *Proceedings of the Twenty-first International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[27] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1), 2005.

[28] I. Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. White paper, Intel Corporation, 2008.

[29] M. Coughlin, A. Ismail, and E. Keller. Apps with hardware: Enabling runtime architectural customization in smart phones. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[30] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *Proceedings of the ACM International Conference on Autonomic Computing (ICAC)*, 2011.

[31] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: memory power estimation and capping. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2010.

[32] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[33] C. Delimitrou and C. Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[34] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 2013.

[35] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[36] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. MultiScale: Memory System DVFS with Multiple Memory Controllers. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2012.

[37] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. MemScale: Active Low-power Modes for Main Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[38] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011.

[39] A. Ebrahim. Dynamic partial reconfiguration management for high performance and reliability in fpgas. *Edinburgh Research Archive*, 2015.

[40] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. Virtualizing and sharing reconfigurable resources in high-performance reconfigurable computing systems. In *High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on*, 2008.

[41] W. El-Essawy, A. P. Ferreira, J. C. Rubio, T. Keller, K. Rajamani, and M. Ware. Enabling real-time data center energy management. 2011.

[42] S. Eyerman and L. Eeckhout. Fine-grained DVFS using on-chip regulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):1, 2011.

[43] S. Fan, S. M. Zahedi, and B. C. Lee. The computational sprinting game. In *ACM SIGOPS Operating Systems Review*, volume 50, pages 561–575. ACM, 2016.

[44] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[45] S. Fang, R. Kanagavelu, B.-S. Lee, C. H. Foh, and K. M. M. Aung. Power-Efficient Virtual Machine Placement and Migration in Data Centers. 2013.

[46] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds:

A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[47] X. Fu, X. Wang, and C. Lefurgy. How much power oversubscription is safe and allowed in data centers. In *Proceedings of the ACM International Conference on Autonomic Computing (ICAC)*, 2011.

[48] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *Proceedings of the International Green Computing Conference and Workshops (IGCC)*, 2011.

[49] A. Gandhi, M. Harchol-Balter, R. Das, J. O. Kephart, and C. Lefurgy. Power capping via forced idleness. 2009.

[50] L. Ganesh, J. Liu, S. Nath, and F. Zhao. Unleash stranded power in data centers with RackPacker. *Workshop on Energy-Efficient Design (WEED)*, 2009.

[51] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[52] A. Goder, A. Spiridonov, and Y. Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, 2015.

[53] W. Godycki, C. Torng, I. Bukreyev, A. Apsel, and C. Batten. Enabling Realistic Fine-Grain Voltage Scaling with Reconfigurable Power Distribution Networks. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[54] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A. Baldini. Statistical profiling-based techniques for effective power provisioning in data centers. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2009.

[55] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar. Benefits and limitations of tapping into stored energy for datacenters. In *ACM SIGARCH Computer Architecture News*, 2011.

[56] S. Govindan, D. Wang, A. Sivasubramaniam, and B. Urgaonkar. Leveraging stored energy for handling power emergencies in aggressively provisioned datacenters. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 75–86. ACM, 2012.

[57] C. Hankendi, S. Reda, and A. K. Coskun. vcap: Adaptive power capping for virtualized servers. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2013.

[58] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[59] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[60] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. Achieving high performance with fpga-based computing. *Computer*, 2007.

[61] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.

[62] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[63] IBM. Ibm and xilinx announce strategic collaboration to accelerate data center applications.

[64] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[65] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[66] S. Jin, J. Cho, X. D. Pham, K. M. Lee, S.-K. Park, M. Kim, and J. W. Jeon. Fpga design and implementation of a real-time stereo vision system. *Circuits and Systems for Video Technology, IEEE Transactions on*, 2010.

[67] D. H. Jones, A. Powell, C.-S. Bouganis, and P. Y. Cheung. Gpu versus fpga for high productivity computing. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010.

[68] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, 2009.

[69] S. Kaxiras and M. Martonosi. Computer architecture techniques for power-efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207, 2008.

[70] S. Kestur, J. D. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In *VLSI (ISVLSI), 2010 IEEE computer society annual symposium on*, 2010.

[71] W. Kim, D. Brooks, et al. A fully-integrated 3-level DC/DC converter for nanosecond-scale DVS with fast shunt regulation. In *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, 2011.

[72] W. Kim, M. Gupta, et al. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[73] I. King. Intel's $16.7 billion altera deal is fueled by data centers.

[74] A. Klimovic, C. Kozyrakis, E. Thereksa, B. John, and S. Kumar. Flash storage disaggregation. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[75] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash? local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 345–359. ACM, 2017.

[76] D. Koch, C. Beckhoff, and J. Teich. Recobus-builder?a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas. In *2008 International Conference on Field Programmable Logic and Applications*, 2008.

[77] T. Kolpe, A. Zhai, and S. S. Sapatnekar. Enabling improved power management in multicore processors through clustered DVFS. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6, 2011.

[78] V. Kontorinis, L. E. Zhang, B. Aksanli, J. Sampson, H. Homayoun, E. Pettis, D. M. Tullsen, and T. S. Rosing. Managing distributed ups energy for effective power capping in data centers. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[79] M. Krips, T. Lammert, and A. Kummert. Fpga implementation of a neural network for a real-time hand tracking system. In *Electronic Design, Test and Applications, 2002. Proceedings. The First IEEE International Workshop on*, 2002.

[80] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[81] G. Lee and R. H. Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *HotCloud*, 2011.

[82] J. Lee and N. S. Kim. Optimizing throughput of power-and thermal-constrained multicore processors using DVFS and per-core power-gating. In *Proceedings of the Annual Design Automation Conference (DAC)*, 2009.

[83] C. Lefurgy, X. Wang, and M. Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2), 2008.

[84] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power management of datacenter workloads using per-core power gating. *Computer Architecture Letters (CAL)*, 8(2), 2009.

[85] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, 2007.

[86] G. Lienhart, A. Kugel, and R. Manner. Using floating-point arithmetic on fpgas to accelerate scientific n-body simulations. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, 2002.

[87] H. Lim, A. Kansal, and J. Liu. Power budgeting for virtualized data centers. In *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, 2011.

[88] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[89] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen. Greencloud: a new architecture for green data center. In *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*, 2009.

[90] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[91] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[92] D. Lo and C. Kozyrakis. Dynamic management of TurboMode in modern multi-core chips. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.

[93] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9, 2008.

[94] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[95] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. The erlangen slot machine: A dynamically reconfigurable fpga-based computer. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 2007.

[96] J. Mars and L. Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[97] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. Soffa. Bubble-up: increasing sensible co-locations for improved utilization in modern warehouse scale computers. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture (MICRO)*, 2011.

[98] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[99] J. Mars, L. Tang, K. Skadron, M. L. Soffa, and R. Hundt. Increasing utilization in modern warehouse-scale computers using bubble-up. *Micro, IEEE*, 2012.

[100] D. McGrath. Will xilinx join the m&a party?

[101] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: eliminating server idle power. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[102] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[103] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[104] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-intensive Services. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[105] D. Meisner, J. Wu, and T. F. Wenisch. BigHouse: A Simulation Infrastructure for Data Center Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2012.

[106] T. N. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu. Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.

[107] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. Snnap: Approximate computing on programmable socs via neural acceleration. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[108] R. Mueller and J. Teubner. Fpgas: a new point in the database design space. In *Proceedings of the 13th International Conference on Extending Database Technology*, 2010.

[109] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2009.

[110] R. Nathuji and K. Schwan. Virtualpower: Coordinated power management in virtualized enterprise systems. In *ACM SIGOPS Operating Systems Review*, 2007.

[111] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[112] A. R. Omondi and J. C. Rajapakse. *FPGA implementations of neural networks*. 2006.

[113] E. Özer, A. P. Nisbet, and D. Gregg. Stochastic bit-width approximation using extreme value theory for customizable processors. In *Compiler Construction*, 2004.

[114] K. Pauwels, M. Tomasi, J. Diaz Alonso, E. Ros, and M. M. Van Hulle. A comparison of fpga and gpu for real-time phase-based optical flow, stereo, and local image features. *Computers, IEEE Transactions on*, 2012.

[115] S. Pelley, D. Meisner, P. Zandevakili, T. F. Wenisch, and J. Underwood. Power routing: dynamic power provisioning in the data center. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[116] F. Peng, N. Ahmed, X. Li, and Y. Lu. Context sensitive stemming for web search. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007.

[117] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang. Octopus-man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.

[118] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[119] N. Pinckney, M. Fojtik, B. Giridhar, D. Sylvester, and D. Blaauw. Shortstop: An on-chip fast supply boosting technique. In *VLSI Circuits (VLSIC), 2013 Symposium on*, pages C290–C291. IEEE, 2013.

[120] D. V. Pryor, M. R. Thistle, and N. Shirazi. Text searching on splash 2. In *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, 1993.

[121] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, 2014.

[122] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. *SIGARCH Comput. Archit. News*.

[123] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No power struggles: Coordinated multi-level power management for the data center. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[124] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level power management for dense blade servers. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[125] S. Reda, R. Cochran, and A. K. Coskun. Adaptive power capping for servers with multithreaded workloads. *IEEE Micro*, 5(32), 2012.

[126] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, 2015.

[127] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's time for low latency. In *Proceedings of the USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2011.

[128] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. *Parallel and Distributed Systems, IEEE Transactions on*, 2014.

[129] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. Fpmr: Mapreduce framework on fpga. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010.

[130] M. Steinbach, L. Ertöz, and V. Kumar. The challenges of clustering high dimensional data. In *New directions in statistical physics*, pages 273–309. Springer, 2004.

[131] B. Subramaniam and W.-c. Feng. Towards energy-proportional computing using subsystem-level power management. *arXiv preprint arXiv:1501.02724*, 2015.

[132] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. Database analytics acceleration using fpgas. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012.

[133] J. Szefer, Y.-Y. Chen, and R. B. Lee. General-purpose fpga platform for efficient encryption and hashing. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, 2010.

[134] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.

[135] L. Tang, J. Mars, and M. L. Soffa. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2012.

[136] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers. In *ACM SIGPLAN Notices*, 2013.

[137] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[138] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *ACM SIGARCH Computer Architecture News*, 2012.

[139] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 2008.

[140] A. Vega, A. Buyuktosunoglu, H. Hanson, P. Bose, and S. Ramani. Crank it up or dial it down: coordinated multiprocessor frequency and folding control. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[141] A. Verma, P. Ahuja, and A. Neogi. pMapper: Power and migration cost aware application placement in virtualized systems. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2008.

[142] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[143] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[144] D. Wang, C. Ren, S. Govindan, A. Sivasubramaniam, B. Urgaonkar, A. Kansal, and K. Vaid. Ace: Abstracting, characterizing and exploiting datacenter power demands. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[145] D. Wang, C. Ren, A. Sivasubramaniam, B. Urgaonkar, and H. Fathy. Energy storage in datacenters: What, where, and how much? In *ACM SIGMETRICS Performance Evaluation Review*, 2012.

[146] G. Wang, D. Anand, et al. Scaling deep trench based eDRAM on SOI to 32nm and Beyond. In *Electron Devices Meeting (IEDM), 2009 IEEE International*, pages 1 –4, dec. 2009.

[147] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[148] X. Wang, M. Chen, C. Lefurgy, and T. W. Keller. Ship: Scalable hierarchical power control for large-scale data centers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[149] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanović. Ramp: Research accelerator for multiple processors. *IEEE Micro*, 2007.

[150] Q. Wu, Q. Deng, Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song. Dynamo: Facebook's data center-wide power management system. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[151] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2005.

[152] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment*, 2013.

[153] J. Yan, Z.-X. Zhao, N.-Y. Xu, X. Jin, L.-T. Zhang, and F.-H. Hsu. Efficient query processing for web search engine with fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 2012.

[154] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[155] J. Yao, X. Liu, W. He, and A. Rahman. Dynamic Control of Electricity Cost with Power Demand Smoothing and Peak Shaving for Distributed Internet Data Centers. 2012.

[156] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ACM SIGARCH Computer Architecture News*, 2000.

[157] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 42(4), 2012.

[158] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. Decoupling Cores, Kernels, and Operating Systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[159] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[160] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[161] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[162] J. Zhu and P. Sutton. Fpga implementations of neural networks–a survey of a decade of progress. In *Field Programmable Logic and Application*. 2003.